Make:
Getting Started with Spark Core and Photon

D7
D6
D5
D4
D3
D2

Early Release
RAW & UNEDITED

Connecting Electronics to the Cloud

Simon Monk

VISIT…

LANZAROTE
Caliente.com

Getting Started with Spark Core and Photon

Simon Monk

MAKER MEDIA
SEBASTOPOL, CA

Getting Started with Spark Core and Photon

by Simon Monk

Copyright (c) 2015 Simon Monk. All rights reserved.

Printed in the United States of America.

Published by Maker Media, Inc., 1160 Battery Street East, San Francisco, CA 94111.

978-1457187018

Contents

Preface . . . . **vii**

1/The Spark Photon **9**
  The Internet of Things **9**
  Sparks in the Clouds **10**
  Other IoT Platforms **12**
    Arduino **12**
    Raspberry Pi and BeagleBone **14**
    Intel Edison **15**
  A Tour of the Spark Photon **15**
  The Spark Core vs. Spark Photon **18**
  Programming **19**
  Summary **20**

2/Quick Start **21**
  Signing Up **21**
  Connecting to WiFi **21**
    Connecting a Core **22**
    Connecting a Photon **28**
  Controlling Pins with the Tinker App **28**
  Project 1. Blink the Tiny Blue LED **30**
  Project 2. Control the Photon’s LED **31**
  Summary **33**

3/Programming the Photon **35**
  The Web IDE **35**
  Coding an App **37**
  Comments **41**
  Variables **42**
  Morse Code **43**
  Flashing SOS **44**
  Functions **48**
  Types **51**

        The int Type. 52
        The float Type. 52
        Other Types. 53
    Arrays. 54
    Loops. 55
    Strings. 57
    Ifs. 58
    Project 3. Morse Code Flasher. 59
        Software. 59
    Summary. 62

4/Breadboard 63
    How Breadboard Works. 65
    Attaching an LED. 66
    Digital Outputs. 67
    Project 4. Morse Flasher (External LED). 68
        Parts. 68
        Hardware. 68
        Software. 69
    Attaching a Switch. 71
    Digital Inputs. 72
    Project 5. Morse Flasher with Switch. 73
        Parts. 73
        Software. 74
        Hardware. 74
        Running the Project. 75
    Analog Outputs. 75
        analogWrite. 77
        An Example. 77
        A Real Analog Output. 79
    Summary. 79

5/The Internet of Things. 81
    Spark Functions. 81
    Project 6. Control an LED Over the Internet. 83
        Software. 84
        Security. 85
        Trying it Out. 89
        Interacting with loop. 90
    Running Functions from a Web Page. 91
    Project 7. Control Relays from a Webpage. 95
        Parts. 98

Design. 100
Construction. 102
Software. 103
Project 8. Morse Code Text Messages. 109
Parts. 110
Software. 110
Hardware. 114
Using the Project. 114
Spark Variables. 115
Analog Inputs. 115
Project 9. Measuring Light over the Internet. 119
Parts. 120
Software. 121
Hardware. 123
Using the Project. 126
Project 10. Measuring Temperature over the Internet. 126
Parts. 127
Software. 128
Hardware. 131
Using the Project. 132
Summary. 134

6/If This Then That. 135
If This Then That. 135
Project 11. Temperature eMail Alerts. 135
Project 12. Ring a Bell for Tweets. 146
Software. 147
IFTTT. 148
Hardware. 149
Using the Project. 150
Project 13. Flash Email as Morse Code. 151
Software. 151
Hardware. 151
IFTTT. 151
Using the Project. 152
Summary. 152

7/Robotics. 155
Project 14. Web Controlled Robot. 155
Parts. 157
Software (Photon). 158
Software (Webpage). 161

        Hardware. 163
        Using the Project. 165
    Summary. 166

8/Machine to Machine Communication. 167
    Publish and Subscribe. 167
    Temperature Monitor Example. 168
    IFTTT and Publish/Subscribe. 172
    Advanced Publish and Subscribe. 173
        Spark.publish. 174
        Spark.subscribe. 174
    Project 15. Magic Rope. 175
        Parts. 177
        Software. 178
        Hardware. 180
        Using the Project. 183
    Summary. 184

9/Advanced Photon. 185
    Configuring a Photon Using USB. 185
    Factory Reset. 188
    Programming a Photon Using Spark Dev. 189
    Debugging with the Serial Monitor. 189
    The Electron. 190
    Power Management. 191
    191
    Summary. 191

A/Parts. 193

B/Photon LED Codes. 197

C/Photon and Core Pinouts. 199

Preface

Early Access

This early access version of *Getting Started with Spark Photon* has been written as the Photon is about to become available, and, so, much of the content is based on the Photon's predecessor the Core, even though the focus of the book is the Photon. The team at Spark have been extremely helpful, sharing information about the minor differences between the Photon and the Core, so that the book anticipates these differences.

When the Photon becomes available, the book will be updated to iron out any inconsistencies between the book and the new device.

1/The Spark Photon

In this chapter, you will learn a little about the Internet of Things in general, as well as the Spark Photon in particular. The Photon and its older brother, the Core, are explored along with some background about where it has come from and where it sits in the pantheon of developer boards.

The Internet of Things

It used to be that the only way you could interact with the Internet was to use a browser on your computer. A browser would allow it to send requests to a web server that would send back information to be displayed.

The browser would display this information using a computer monitor, and the “user” would type text on his or her keyboard and follow hyperlinks with the click of a mouse. As far as inputs and outputs were concerned, those were your options.

The Internet of Things (abbreviated to IoT) has changed all this. Now all sorts of sensors and appliances can be connected to the Internet. The IoT encompasses a wide range of systems including:

- Home automation systems where lighting, heating and doors are controlled using web browser or network-enabled smartphone applications. These may be used to control systems over the local area network, or over the Internet using WiFi or a cellular network
- Arrays of sensors, such as the Safecast open radiation monitoring system that was developed following the Fukushima nuclear disaster

Products and maker projects that will become part of the IoT are springing up all over the place. These include successful projects like the Nest smart thermostat as well as many IoT products that use the accelerometer, location services and communication features of smartphones to capture information about people's health and activity levels.

Since so many people are creating IoT projects, it makes perfect sense to provide a simple modular framework for both hardware and software that provides an easy-to-use IoT technology kit. This is exactly where the Spark team come in. They provide IoT technology in a box. In a very small and low cost box. What is more, the technology is easy to use, open source and based on the very popular Arduino software framework.

Sparks in the Clouds

The hardware component of this IoT framework is the Spark Photon. The Photon is the next generation of Spark's IoT platform that began with the Spark Core. The Spark Photon is backwards compatible with the Spark Core, and so much of what is detailed in this book about the Photon will also work with the older Core.

Although other technologies exist to help you build IoT devices, they often neglect the all-important software framework that allows the device to communicate with other devices and browsers over the Internet. The Spark approach, by contrast, integrates the hardware and software seamlessly.

Figure 1-1 shows how a typical IoT device built using a Spark Photon or Core might interact with the Internet.

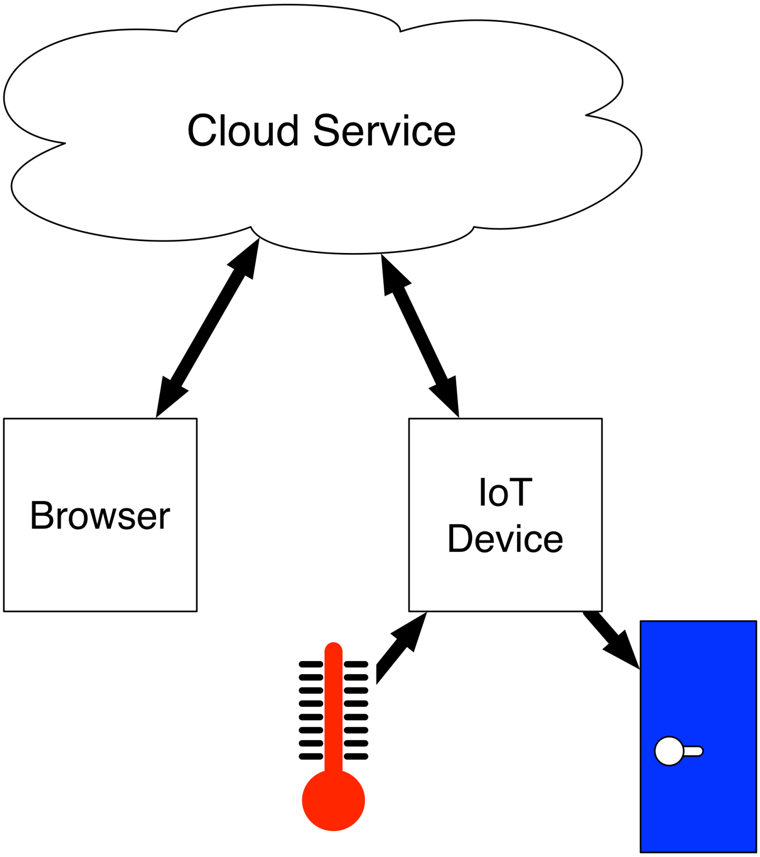


Figure 1-1. *Figure 1-1. Internet of Things Communication*

An IoT device using a Photon/Core might provide remote unlocking of a door. In such a case, a user would access a web page on their browser that has an “Unlock” button. This page will have been served from a web server somewhere on the Internet. When the user clicks on the Unlock button in their browser, a message is sent from the browser to the IoT device via a cloud service that is aware of the Photon running the IoT device. The message is then passed to the Photon/Core that controls an electromechanical door latch to unlock the door.

If, on the other hand, the IoT device was acting as a sensor, lets say for temperature, then the Photon/Core could send temperature readings to a cloud service. Those readings could be stored temporarily until the user's browser has a chance to pick them up and display the latest reading on the browser.

To use Spark's cloud service, you first register with Spark and then identify each of your Photons/Cores, which will have registered themselves with the cloud service as being yours. All the Photon needs to do to register itself is to have access to your WiFi network. This process not only allows you to ensure that you know which Photon or Core you are interacting with, but also allows you to program your Photons and Cores over the air from the comfort of your web browser.

Other IoT Platforms

Before plunging into those delightfully warm and pleasant waters of the Spark Photon pool, its is worth exploring some of the Photon's competitors. This will also reveal something of the motivations behind the design of the Photon.

The Spark Photon is of course not the only IoT device around. In fact the single most used platform for IoT development is the Arduino microcontroller board, although the Raspberry Pi single-board computer is also used extensively in IoT projects.

Arduino

Microcontrollers are essentially low-powered computers on a chip. They have input/output (I/O) pins to which you can attach electronics so that the microcontroller can - well - *control* things. The Arduino is a simple-to-use and low-cost, ready-made board that allows you to make use of a microcontroller in your projects.

The Arduino has become the platform of choice for makers and hackers looking for a microcontroller to use, and the most common Arduino model is the Arduino Uno.

The popularity of Arduino is due to many factors, including:

- Low cost (around USD 25 for an Arduino Uno)
- Open source hardware design - there are no secrets to its design and built-in software

- Easy to use Integrated Development Environment with which to program the Arduino
- Plug in shields that plug onto the top of the Arduino and add features such as displays and motor drivers

There is of course one factor that makes an Arduino Uno by itself useless as an IoT device, and that is that it has no network connection, either wired or wireless. This means that you either need to use one of the specialized Arduino models that include an Ethernet network port (such as the Arduino Ethernet) or add a WiFi or Ethernet shield to the Arduino that then gives it the network connection it needs to communicate over the Internet. This adds considerably to the size and cost of your project.

Figure 1-2 shows an Arduino Uno with a WiFi shield attached. The total cost of this combination is over USD 100.

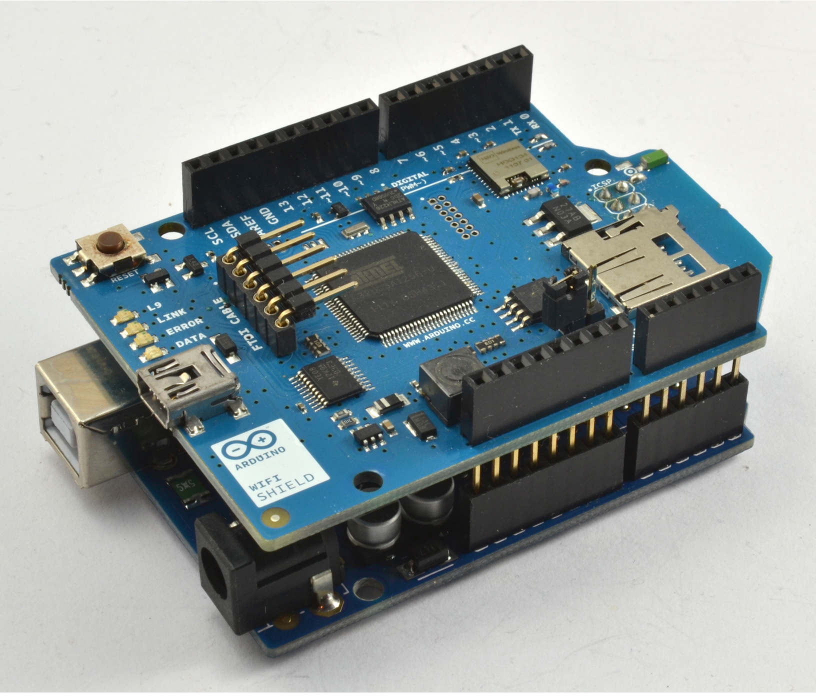


Figure 1-2. *Figure 1-2. An Arduino Uno with WiFi Shield*

Another possibility is to use the Arduino Yun. This device is the same size as an Arduino Uno but it has a built-in WiFi module.

On the face of it, this provides similar hardware capabilities to the Spark Photon, but at a much higher price of around USD 75.

All of these Arduino-based solutions suffer from one major disadvantage as an IoT platform, and that is software. They provide the base capabilities to communicate with the Internet but do not offer any software framework to make it easy to create IoT projects without a lot of tricky network programming.

Later, you will see how the Spark Photon borrows many of the concepts of Arduino, including its programming language, but then provides a software framework with which to build your IoT projects, all at much lower cost than Arduino can compete with.

Raspberry Pi and BeagleBone

The Raspberry Pi and BeagleBone Black (Figure 1-3) are both single-board computers, about the size of a credit card, that run the Linux operating system. They have USB ports and HDMI video output, so you can set them up with a keyboard, mouse and monitor and use them just like a regular computer.

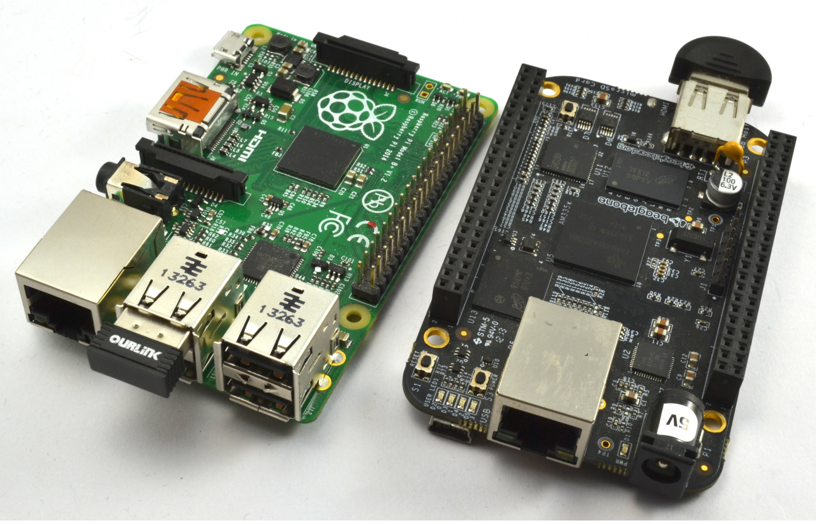


Figure 1-3. *Figure 1-3. A Raspberry Pi and BeagleBone Black*

The Raspberry Pi is shown on the left of Figure 1-3 and the BeagleBone Black on the right. Both boards can use low-cost USB

WiFi adaptors and have I/O pins to control electronics an interface with sensors making them quite suitable for IoT projects.

Although both boards are quite low cost, the Raspberry Pi from USD 25 and the BeagleBone from USD 55, they are quite large (compared to a Photon) and generally contain a lot more than you need for a simple IoT project.

Intel Edison

Intel has developed a small Linux-based board called the Edison. The Edison is designed to be embedded into IoT projects and is perhaps the most direct competition to the Spark Photon.

The Edison is small but is considerably more expensive than the Photon. It has a delicate 70-contact connector that requires a separate breakout board if you want to start attaching external electronics to it. There are several such prototyping boards available, the most popular of which is an Arduino-compatible board.

Although receiving a lot of interest in the Maker community, this is probably a board that will lend itself best to high-end or professional use, not least because the device is a lot harder to get started with.

A Tour of the Spark Photon

Figure 1-4 shows a Spark Photon with some of its main features labelled.

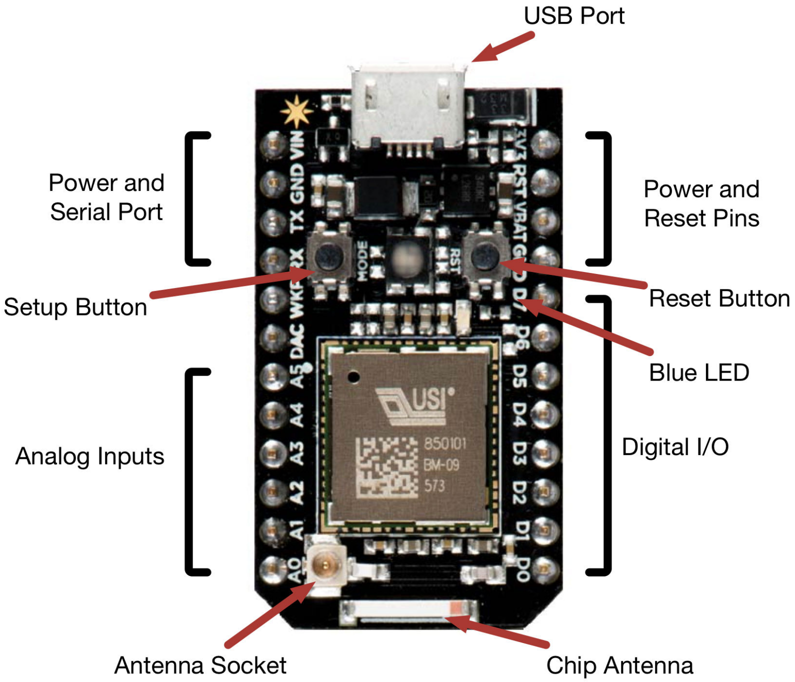

USB Port
Power and Serial Port
Power and Reset Pins
Setup Button
Reset Button
Blue LED
Analog Inputs
Digital I/O
Antenna Socket
Chip Antenna

Figure 1-4. *Figure 1-4. The Spark Photon*

The Photon has two buttons. The Reset button resets the Photon, restarting the processor. The Setup button ("Mode" button on the Core) is used in conjunction with the Reset button to allow more severe resetting of the Photon, including a full factory reset.

At the top of the board, you will find the micro-USB port. Although the main point of this is to provide power to the Photon, it can also be used for USB programming of the board (see Chapter 9) and for USB communication with a computer.

Moving around the board clockwise, there is a section of pins labelled "Power and Reset" on Figure 1-4. The top-most pin (3.3V) is the 3.3V power used by the Photon, as all the logic operates at 3.3V rather than the 5V that you might be used to as an Arduino user. The RST pin allows you to achieve the same effect as pressing the reset button, put electrically, by connecting the pin to ground. You are fairly unlikely ever to want to use this pin.

The VBAT pin allows a small backup battery (or supercapacitor) to be attached to the Photon, to power it while it is in deep sleep mode.

The pins labelled D0 to D7 are general purpose GPIO pins that can act as either digital inputs or outputs (see chapter 4). Some of these pins can also act as analog outputs (pins D0 to D4) using a technique called PWM (Pulse Width Modulation). There is a blue LED next to pin D7 that is connected to D7. You can turn this on and off from your programs, or using the Tinker app on your smartphone.

The Photon has a built-in chip antenna that will work fine in most WiFi situations, but the Photon also has a tiny antenna socket to which an external antenna can be attached. This is useful for extending the WiFi range of the device by adding a more sensitive or directional antenna.

The pins A0 to A5 are primarily analog inputs which can measure a voltage between 0 and 3.3V. These will typically be used for sensors. For example, in Chapter 6 you will use an light sensor with one of these pins. The analog pins can also be used as digital inputs or outputs just like the pins D0 to D7 and, like the digital pins, some of the analog pins (A4, A5) can also be used as PWM analog outputs.

The DAC pin is a special analog output pin. This is a true analog output that can be set to any voltage between 0 and 3.3V with 12 bits of resolution.

The WKP pin is used to wake up the Photon after it has put itself into a deep sleep mode. Writing software that allows the Photon to sleep some of the time makes it practical to run the Photon from batteries where power consumption needs to be minimized.

The TX and RX (Transmit and Receive) are used for serial communication. Above these pins is a second GND pin and the VIN pin. You can supply between 3.6V and 6V to the VIN pin to power the board as an alternative to using the USB port.

The Spark Core vs. Spark Photon

The Photon is not the first IoT board to be produced by Spark. That honor belongs to the Spark Core. Both boards are shown side by side in Figure 1-5, and as you can see the boards look very similar to each other.

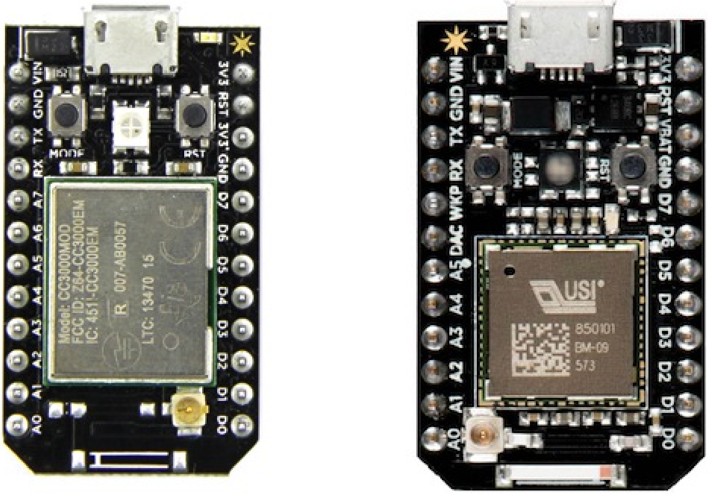

VIN GND TX RX A7 A6 A5 A4 A3 A2 A1 A0
3V3 RST 3V3* GND D7 D6 D5 D4 D3 D2 D1 D0
MODE
RST
Model: CC3000MOD
FCC ID: Z64-CC3000EM
IC: 4511-CC3000EM
007-AB0057
LTC: 13470 15
VIN GND TX RX WKP DAC A5 A4 A3 A2 A1 A0
3V3 RST VBAT GND D7 D6 D5 D4 D3 D2 D1 D0
MODE
RST
USI
850101
BM-09
573

Figure 1-5. *Figure 1-5. The Spark Core (left) and Photon (right)*

The main differences are that the Photon uses a different WiFi chip (Broadcom rather than Texas Instruments). It also uses a faster processor with more RAM.

The pins are mostly the same between the two boards, however there are a few important changes to be aware of. First, the Core does not have DAC and WKP pins; instead it has two extra analog inputs, A6 and A7.

The examples in this book will have all been tested on both the Photon and the Core, so if you have one of the older Cores you should be just fine.

Programming

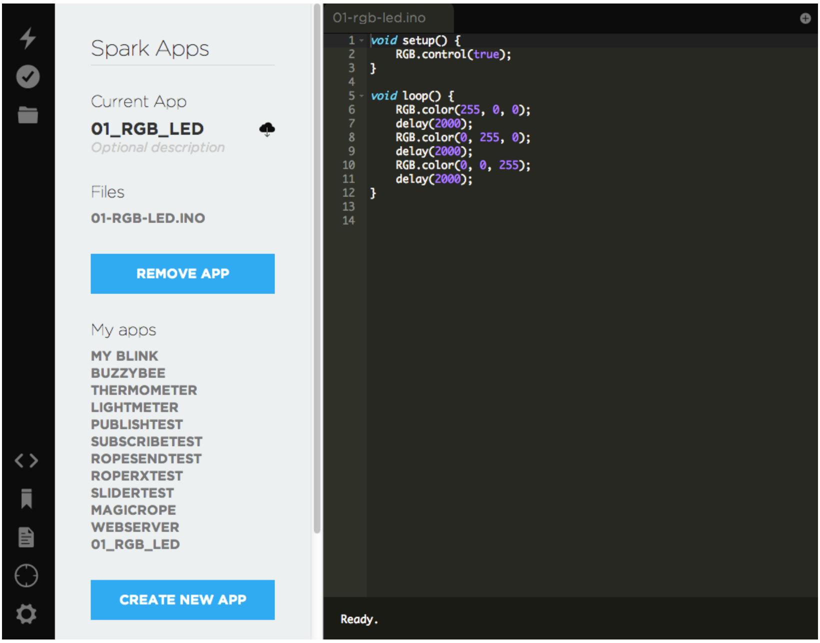
Spark has gone to great efforts to make programming the Photon and Core as easy as possible, both by providing a very simple-to-use development environment and also by choosing the Arduino C programming language as the basis for the Photon's programming language.

Since the Photon is an Internet device, it makes perfect sense to program the device over the Internet, so most of the time, you will write the code for your Photon in your web browser and then push it out to your Photon, which will be looking out for updates. An offline programming environment is also available for more advanced users (see Chapter 9).

For those thinking of using the Photon for commercial applications, the ability to update over-the-air is incredibly powerful, as it allows remote updates of the Photon's software from anywhere in the world.

This may start a few alarm bells ringing. After all if the Photon was controlling your home's front door or heating, you would not want someone updating your cores and telling them to unlock your doors and turn the heating up to maximum. Fortunately, there are several security mechanisms that are used to secure the Photon. This includes the use of secure socket communication as well as authentication keys that ensure that only you can update your cores.

Figure 1-6 shows the Web IDE (Integrated Development Environment) for the Spark Photon/Core.


Spark Apps
Current App
01_RGB_LED
Optional description
Files
01-RGB-LED.INO
REMOVE APP
My apps
MY BLINK
BUZZYBEE
THERMOMETER
LIGHTMETER
PUBLISHTEST
SUBSCRIBETEST
ROPESENDTEST
ROPERXTEST
SLIDERTEST
MAGICROPE
WEBSERVER
01_RGB_LED
CREATE NEW APP
01-rgb-led.ino

```
void setup() {
    RGB.control(true);
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
    delay(2000);
}
```

```
Ready.
```

Figure 1-6. *Figure 1-6. The Spark Web IDE*

The Web IDE allows you to write code that you can then send down to one of your Photons over the internet.

Summary

By now, you must be itching to get started with the Spark Photon, so in Chapter 2, you will learn how to get up and running with your Photon and install your first program onto it.

2/Quick Start

In this chapter you will launch right into using the Spark Photon or Core. If you are new to programming and this kind of technology, then just work through the instructions here and do not worry too much if there are things that you don't understand, these will all be explained in much more detail later in the book.

Signing Up

One of the things that you get when you buy a Spark device is access to Spark's Cloud Service. You will need to register to gain access to this. Registering with the cloud service is essential as it allows you to manage your Photons and Cores over the Internet, including programming them remotely.

The cloud service is also what a Photon/Core uses to communicate with other Photons, Cores and browser applications.

Your first step should therefore be to sign up at www.spark.io. Click on the LAUNCH button to begin the sign up process. Don't worry, this is strictly just to use your Photon over the Internet, you will not be bombarded with spam.

Once you are all signed up, you will find yourself in the Web IDE ready to start writing programs. Before you can do that, you need to connect your Photon to the Internet.

Connecting to WiFi

Although it is possible to program a Photon or Core over its USB connection (see Chapter 9), it makes more sense to program an Internet device like this over the Internet using the Web IDE. To do that, the Photon or Core needs to be connected to the Internet over WiFi. To be able to connect using WiFi, it needs to know the SSID (network name) and passphrase of your home network.

As you may have noticed, the Core and Photon have no keyboard or display, so you need a way of telling them these details.

Connecting a Core

To help you get your Core connected, Spark has produced a smartphone app available for Android and iPhone. This app (TinkerApp) actually does more than allow you to connect your Core to the Internet; it also allows you to control the Core, turning pins on and off, etc. If you do not have a suitable smartphone, then all is not lost; you can configure both the Core and Photon using the USB cable (see Chapter 9).

When you plug in a brand-new Core, it should blink dark blue, indicating that it is "listening" for a WiFi network to connect to. Unless it is in this state, you will not be able to connect it to your wireless network. You can put your device back into this state by doing a factory reset on it as described in Chapter 9.

You will need an Android or iOS (iPhone/iPad) device to use the Tinker app to connect your device, and the phone or tablet must be connected to the WiFi network that you want to join the Core to.

When you first start the Tinker app you will be prompted to log into your Spark account (Figure 2-1). Sign in using the same email and password that you gave when you signed up to Spark.

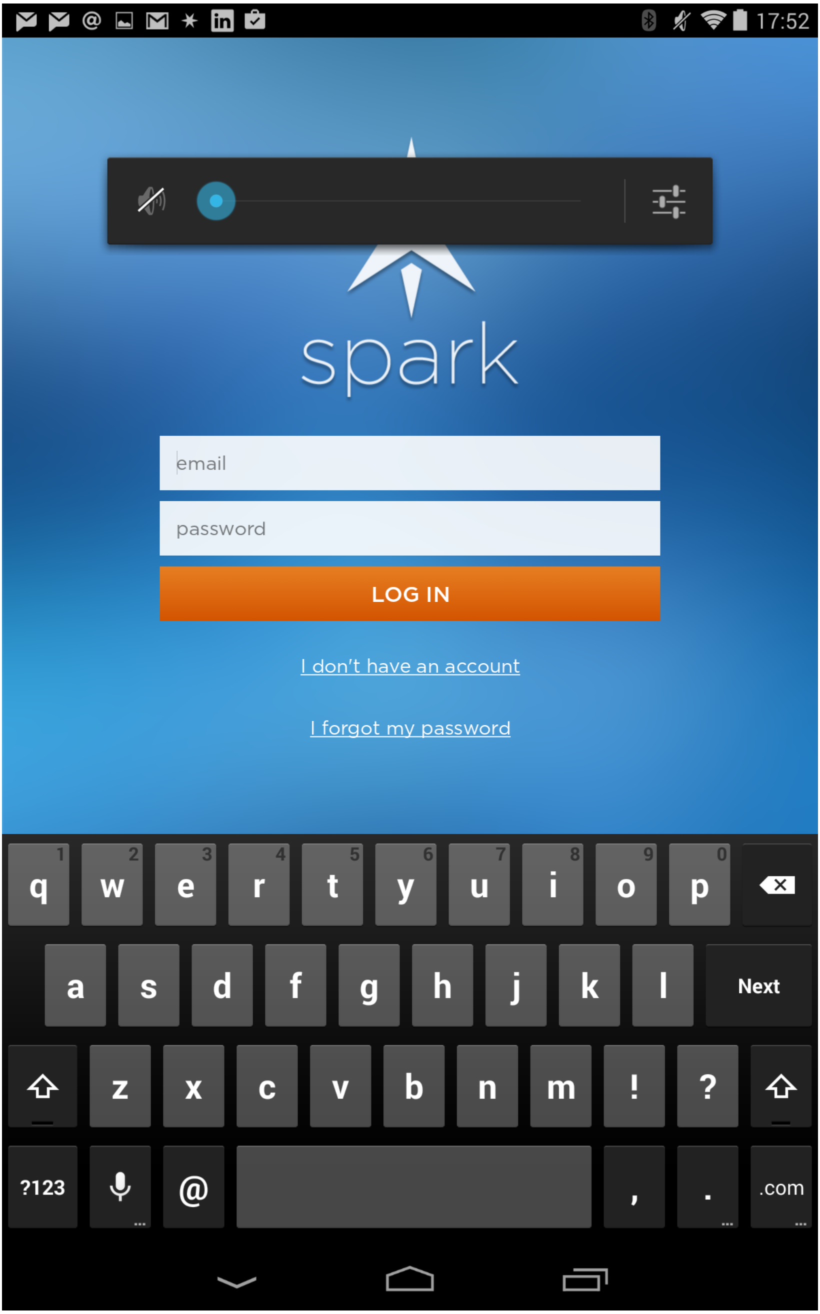
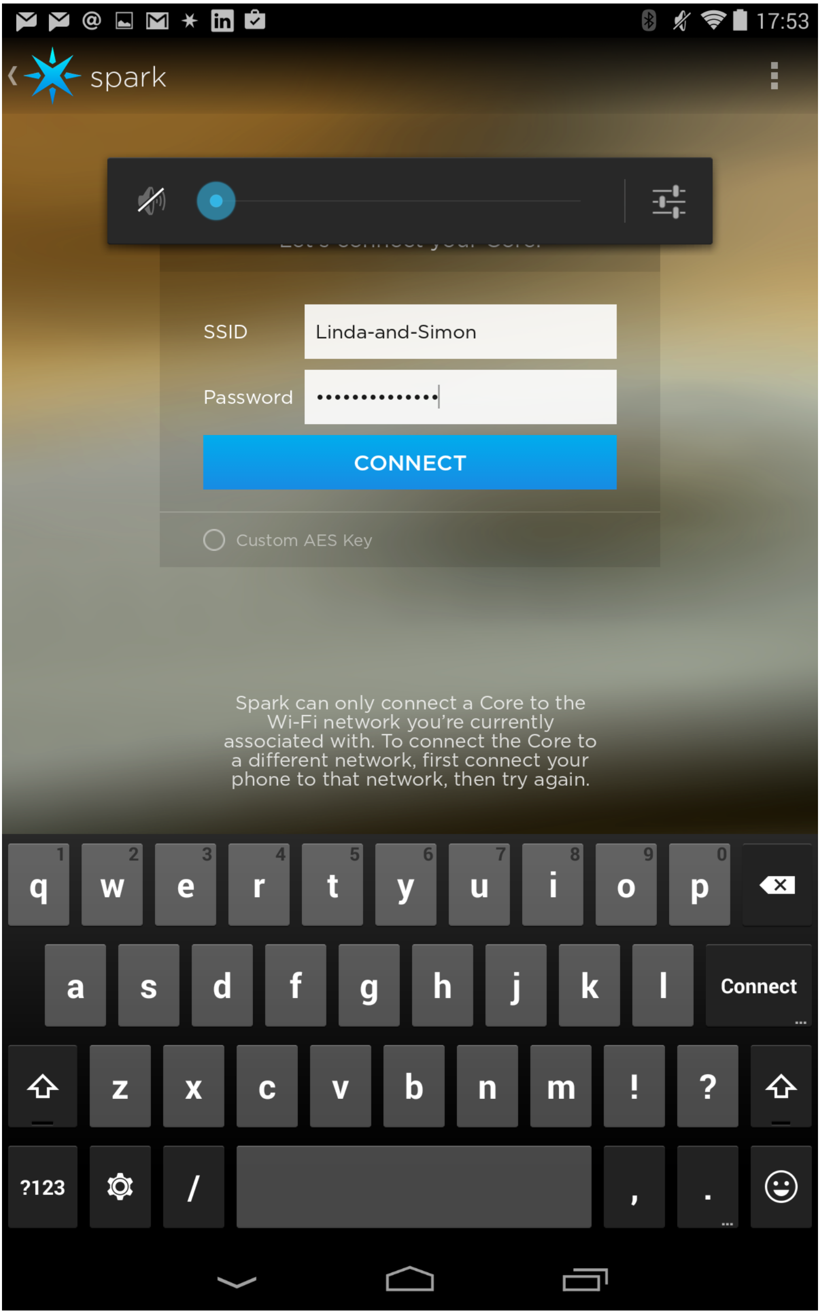

17:52
spark
email
password
LOG IN
I don't have an account
I forgot my password
Next
?123
.com

Figure 2-1. *Figure 2-1. Logging into your Spark Account*

After you are logged in, you will be prompted to enter your WiFi password. The SSID (wireless network name) will be pre-filled as your device is already connected to that wireless network (Figure 2-2).


17:53
spark
CONNECT
○ Custom AES Key
Connect
?123

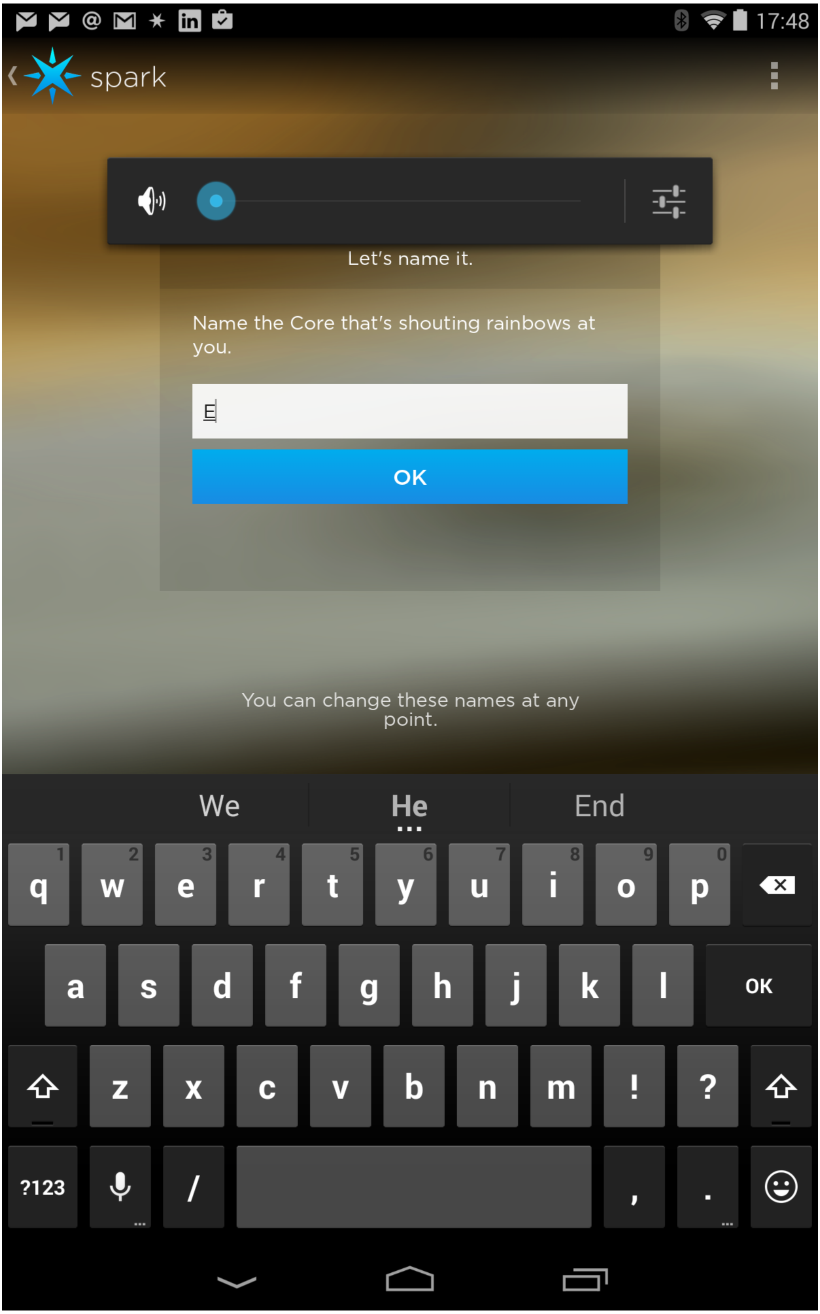
SSID: Linda-and-Simon

Password: ••••••••••••••

Spark can only connect a Core to the Wi-Fi network you're currently associated with. To connect the Core to a different network, first connect your phone to that network, then try again.

Figure 2-2. *Figure 2-2. Entering your WiFi Password*

The app should then discover your Core and prompt you to supply a name for your Core (Figure 2-3). At this point the Core will be “blinking rainbows at you” as it says in the dialog.


17:48
spark
We
He
End
OK
?123

Let's name it.

Name the Core that's shouting rainbows at you.

E

You can change these names at any point.

Figure 2-3. *Figure 2-3. Naming your Device.*

The Core should then reset, and when its finished the LED should be gently breathing in cyan and ready to start using.

Connecting a Photon

The documentation for connecting a Photon is not yet available. It may be exactly the same as the Core and use the Tinker App. But there is also talk of a SoftAP (Soft Access Point) method of connecting that may mean that the Photon temporarily hosts a wireless access point, to which you can connect and then enter credentials for your normal wireless network. Please check on www.spark.io for the latest information.

Controlling Pins with the Tinker App

Before trying out our first program, we can use the smartphone Tinker app to control the Photon or Core. This is the same smartphone app that you used to configure your Photon/Core, but having configured the device, you can use the same app to interact with it.

In this case, you will use it to turn the built-in blue LED attached to pin D7 on and off.

The first step is to launch the app on your Android phone or iPhone. Log in to Spark if you are prompted and then you should find yourself at the main control screen.

Before you can turn pin D7 on and off, you first have to tell the Photon/Core that pin D7 is to be used as a digital output, so tap and hold the screen next to D7 and the menu shown in Figure 2-4. Tinker should appear.

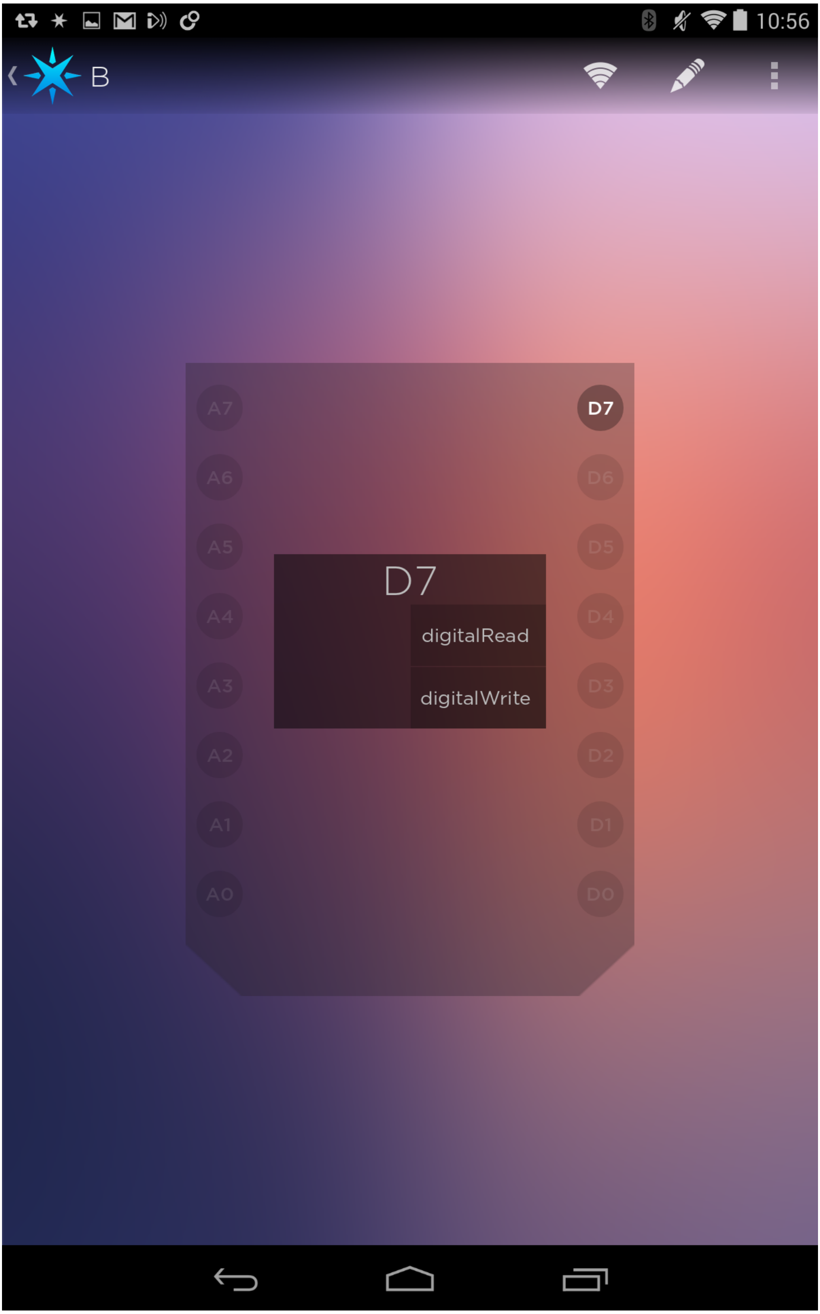

10:56
B
A7
A6
A5
A4
A3
A2
A1
A0
D7
D6
D5
D4
D3
D2
D1
D0
D7
digitalRead
digitalWrite

Figure 2-4. *Figure 2-4. Controlling D7 with the Tinker App*

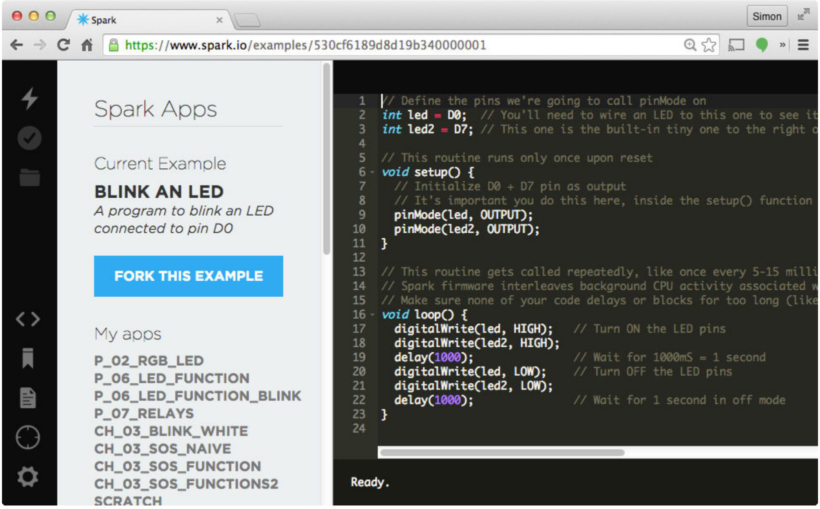
Select the option digitalWrite. Now, if you tap on the D7 pin on the screen, the blue LED next to pin 7 will toggle on and off.

Project 1. Blink the Tiny Blue LED

Programmers have a tradition that the first program that they write in any language should be called "Hello World". This usually involves just getting the program to display the words "Hello World". Clearly the Photon/Core doesn't have anywhere to display those words, so we will have to think of something else as our easiest possible program.

Arduino and other similar boards have adopted the task of blinking an LED as the most basic type of thing that you might want to do with a board like the Photon, so lets start with that.

Log in to spark.io by clicking on the LAUNCH button and you should find yourself at the Web IDE page. At the bottom-left of the screen, you will find a section called Example Apps. Click on the option BLINK AN LED and the program shown in Figure 2-5 will load up into the IDEs editor.


Spark
Simon
https://www.spark.io/examples/530cf6189d8d19b340000001
Spark Apps
Current Example
BLINK AN LED
FORK THIS EXAMPLE
My apps
P_02_RGB_LED
P_06_LED_FUNCTION
P_06_LED_FUNCTION_BLINK
P_07_RELAYS
CH_03_BLINK_WHITE
CH_03_SOS_NAIVE
CH_03_SOS_FUNCTION
CH_03_SOS_FUNCTIONS2
SCRATCH

A program to blink an LED connected to pin D0

```
// Define the pins we're going to call pinMode on
int led = D0;  // You'll need to wire an LED to this one to see it
int led2 = D7; // This one is the built-in tiny one to the right

// This routine runs only once upon reset
void setup() {
  // Initialize D0 + D7 pin as output
  // It's important you do this here, inside the setup() function
  pinMode(led, OUTPUT);
  pinMode(led2, OUTPUT);
}

// This routine gets called repeatedly, like once every 5-15 milli
// Spark firmware interleaves background CPU activity associated
// Make sure none of your code delays or blocks for too long (like
void loop() {
  digitalWrite(led, HIGH);   // Turn ON the LED pins
  digitalWrite(led2, HIGH);
  delay(1000);               // Wait for 1000mS = 1 second
  digitalWrite(led, LOW);    // Turn OFF the LED pins
  digitalWrite(led2, LOW);
  delay(1000);               // Wait for 1 second in off mode
}

```

```
Ready.
```

Figure 2-5. *Figure 2-5. The Web IDE with Blink Loaded*

Do not worry about how this program works, all will be explained in the next chapter, for now we just want to see that LED blink, so click on the icon in the top left of the IDE that looks like a lightning bolt. This is the “Flash” button that will flash the blink program onto your Photon.

You should see a message appear in the status area at the bottom of the screen that says: *Flash successful! Please wait a moment while your Photon is updated…*

Occasionally, the flashing will appear to hang and nothing happens. Eventually, the IDE will time-out and you can try again. It may help to reset the Photon/Core by clicking on the Photon’s reset button.

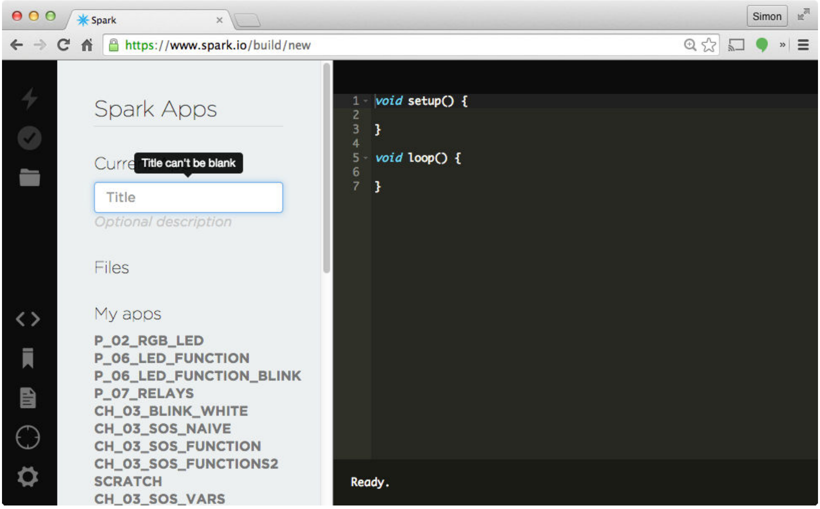
At this point, the RGB LED on the device should start blinking away in purple, indicating that the Photon/Core is being updated. After the device has received the program, it will reset itself and the RGB LED will blink green for a little while. Once the Photon/Core restarts, the tiny blue LED next to pin 7 will flash on and off every two seconds.

You will meet this program in the next chapter, where all will be explained, but before that, in Project 2, you will learn how to create a new program and type in the code for it.

Project 2. Control the Photon’s LED

Like the tiny blue LED attached to pin D7 of the Raspberry Pi, you can also control the RGB LED that the Photon uses to indicate its status. In this mini-project, you will type a short program into the Web IDE that will make the RGB LED light first red, then change to green, then after a further two seconds turn blue and so on.

The first step is to click on the CREATE NEW APP button on the left hand side of the Web IDE. This will start a new app as shown in Figure 2-6.


Spark
https://www.spark.io/build/new
Simon
Spark Apps
Curre
Title
Optional description
Files
My apps
P_02_RGB_LED
P_06_LED_FUNCTION
P_06_LED_FUNCTION_BLINK
P_07_RELAYS
CH_03_BLINK_WHITE
CH_03_SOS_NAIVE
CH_03_SOS_FUNCTION
CH_03_SOS_FUNCTIONS2
SCRATCH
CH_03_SOS_VARS

Title can't be blank

```
void setup() {

}

void loop() {

}
```

Ready.

Figure 2-6. *Figure 2-6. Creating a New App.*

As you can see from Figure 2-new-app1 you are being prompted for a name for the app. This needs to be unique and whatever you type here as a name will be converted into uppercase. If you want to separate the words of the app name, then you can use an _ character. Call the app "RGB_LED."

When you create a new project, the Web IDE will automatically create a blank template into which you can write your code. At the moment, the program for the app looks like this:

```
void setup() {
}
void loop() {
}
```

Change this text so that it appears as below:

```
void setup() {
    RGB.control(true);
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
```

```
        delay(2000);
    }
```

All the text that you are going to add into the template needs to be tabbed in one tab space, so on each new line, press the TAB key once before typing the line of code. Notice that each line of code must end in a semi-colon.

When you have typed in all of the code, save the app by clicking on the Save icon (looks like a folder) and then hit the flash button (lightning bolt) to program the Photon.

It does not matter whether the Photon is plugged into the breadboard or not. However, keeping your Photon on the breadboard will protect its pins against getting bent or accidentally shorting together if they touch something metal.

Error Messages

If you mistyped something then you will see an error message. A common mistake is a missing semi-colon on the end of a line. If this is your error, then if you look carefully at the error message, there is a lot of incomprehensible stuff before you get to the line that says:

```
the_user_app.cpp:3:1: error: expected ';' before '}' token
```

This is just saying that there is a semi-colon missing somewhere. So go back and check over your code carefully.

Once the Photon/Core has finished flashing purple and has restarted, you should see the RGB LED cycle through red, green and blue.

Summary

Congratulations, you have made your Photon or Core do something!

In the next chapter you will revisit both the apps that you have run on your core to understand a bit more about how they work and how to go about writing your own apps.

3/Programming the Photon

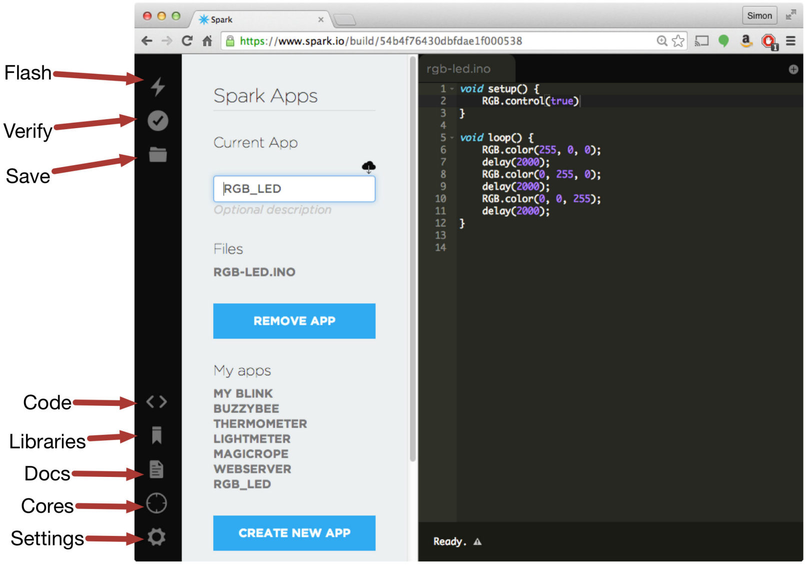
Chapter 2 gave you your first taste of programming the Photon/Core, this chapter will take you deeper into the world of programming, and, if you are new to coding, teach you some of the basics of the C language that these devices use.

If you are a seasoned programmer, you can skip big chunks of this chapter.

The Web IDE

In chapter 2, you touched on using the Web IDE.

Apart from typing in the code and clicking on a couple of the buttons, there are large uncharted areas of the Web IDE to explore. Figure 3-1 shows the Web IDE with labels for some of its buttons.


Flash
Verify
Save
Code
Libraries
Docs
Cores
Settings
Spark
https://www.spark.io/build/54b4f76430dbfdae1f000538
Simon
Spark Apps
Current App
RGB_LED
Optional description
Files
RGB-LED.INO
REMOVE APP
My apps
MY BLINK
BUZZYBEE
THERMOMETER
LIGHTMETER
MAGICROPE
WEBSERVER
RGB_LED
CREATE NEW APP
rgb-led.ino
Ready.

```
void setup() {
    RGB.control(true)
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
    delay(2000);
}
```

Figure 3-1. *Figure 3-1. The Web IDE*

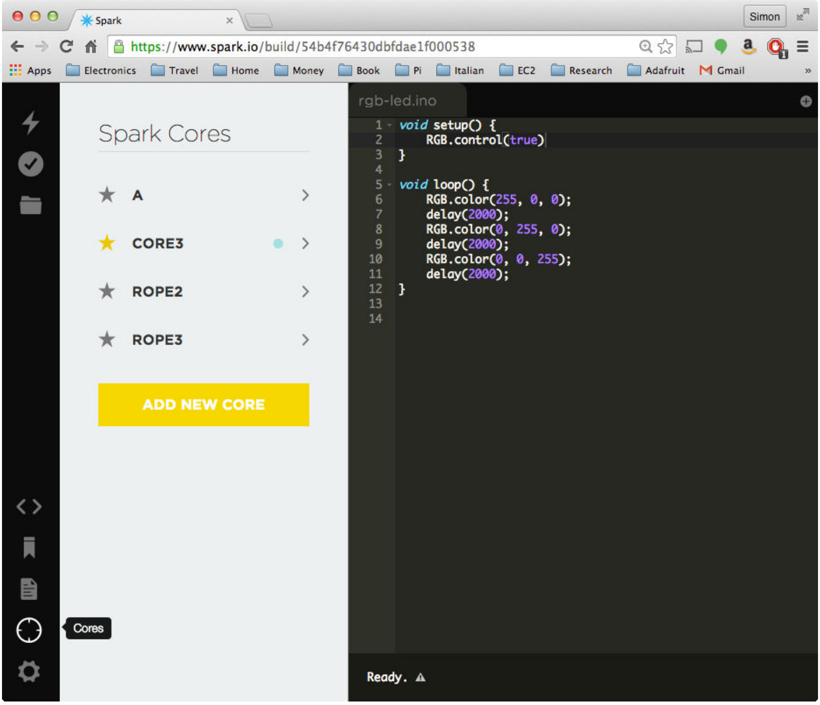
You have already used the Flash button at the top of the button panel. The Verify button will check a program and make sure the code is legal without actually sending it to the Photon. Since the Flash button also verifies the program and will not send it unless its legal, you may find the Verify button to be of limited use.

The Save button will save whatever app is currently showing in the editor window to the right. The <> Code button toggles the side panel that contains your list of apps and the buttons that allows you to remove an app or create a new app, as you did in Chapter 2.

Underneath the Code button, the Libraries button allows you access to a huge body of community-contributed libraries of code that you can make use of in your apps. Many of these libraries are concerned with interfacing specific types of hardware such as sensors and displays.

The Docs button opens up the web page for the documentation for the Photon.

The Photons button allows you to manage your Photons and Cores. When you click on it a side panel slides out, listing all the devices associated with your account so that you can manage them in various ways.


Spark
Simon
https://www.spark.io/build/54b4f76430dbfdae1f000538
Apps
Electronics
Travel
Home
Money
Book
Pi
Italian
EC2
Research
Adafruit
Gmail
Spark Cores
A
CORE3
ROPE2
ROPE3
ADD NEW CORE
Cores
rgb-led.ino
Ready.

```
void setup() {
    RGB.control(true)
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
    delay(2000);
}
```

Figure 3-2. *Figure 3-2. Managing Your Photons*

The list will show all the Photons and older Cores you have that are associated with your account. The gold star next to the device indicates that this is the currently selected device, so if you clicked on the flash button, it is that device that would be programmed.

The blue dot to the right of the device name indicates that the device is active and connected to the cloud service.

Coding an App

Lets start by dissecting the app that you wrote in Project 2 that changed the color displayed on the Photon's built-in LED between red, green and blue at two-second intervals. As a reminder, the code for this is listed below:

```
void setup() {
    RGB.control(true);
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
    delay(2000);
}
```

A program (or "app" in Spark terminology) takes the form of a series of instructions that the computer (in this case Photon or Core) must execute one after the other. There is a little more to it than this, or the program above wouldn't have strange *{* and *}* symbols as well as mystical words like *void*, *setup* and *loop* and a seemingly random assortment of punctuation.

As you can see, the code is in two sections. One section that starts with the words *void setup* and another that starts with *void loop*. Each of these sections of the program code are called a *function*. A function is a way of grouping together a set of instructions and giving them a name.

Looking at the first function, it begins with the word *void*. This might seem an unusual word to indicate the start of a function. In case you are interested, this is because every function has to specify a *type* (more on types later). Both the functions in this program have no type, so we have to indicate this fact using the word *void*. One thing that you will notice right away about programming languages is that they are really fussy. Computer scientists love things to be orderly and precise, hence we need to write *void* before our setup and loop functions.

After the word *void* comes the function name *setup* followed by *()*. Later on, you will see functions that have something inside the *()* called parameters. Again, the precise nature of the C language dictates that even if this function happens not to have any parameters, you must still include the parentheses.

Every app must include the functions *setup* and *loop*. That is why when you click on the button CREATE NEW APP the skeleton code for these two functions will appear as below, ready for you to add in your own instructions.

```
void setup() {
}

void loop() {
}
```

The program code that you add to a function needs to go between the { and }, with a ; after each instruction line. Over the years, programmers have developed conventions on how you should write your code to make it as easy as possible for another programmer to understand the code, or for you to understand it when you return to it after not looking at it for a while.

These conventions include indenting each of the lines inside the function, so that it is easy to see that they belong to it. You will hear such sections of indented code being referred to as a *block* of code. Unlike the Python language the C language used by the Photon and Core ignores all indentation. If you leave out the indentation, the program will still work just fine, its just a matter of convention to keep things neat.

The use of the two functions *setup* and *loop* is borrowed from Arduino and works in just the same way. When your device is first powered up, or after you press the Reset button, the lines of code inside the *setup* function will be “run” (executed) just once. They will not be run again until the Photon or Core is reset.

In the case of our example program, the setup function has just a single line of code in it:

```
RGB.control(true);
```

This command tells the Photon/Core that we want to take control the color of its RGB LED rather than let it do all its fancy blinking to tell us its status. Don’t worry, the device will revert to blinking its status colors when we upload a different app that does not take control of the RGB LED. We will still be able to send new apps the the Photon/Core, its just we will not know whether it is connected properly to the Internet or not with the app running.

The *loop* function is different to the *setup* function, because it will run over and over again. As soon as it has finished running the last of the commands within its curly braces, it will start running the first one again.

In this case, there are four lines of code that will be run each time around the loop. The first line is:

```
RGB.color(255, 0, 0);
```

Our *setup* function code gave us control of the RGB LED. Now we need to instruct it to change color. The first line of the loop code sets the color to be red. It does this because *RGB.color* is a function like *setup* and *loop*, but in this case, a function that is built-in to the Photon's software which we can just make use of. Where the *setup* and *loop* functions had empty *()* after them, you can see that *RGB.color* has three parameters, separated by commas. These three parameters represent the brightness of the red, green and blue channels of the LED respectively. The number supplied for each parameter is between 0 and 255, where 0 means no light and 255 means full brightness. So, the values of 255, 0, 0 mean full brightness red, no green and no blue. If we were to use 255, 255, 255 the LED would shine all three colors at full brightness and appear white.

Once the color is set to red, we want the app to do nothing for a couple of seconds before we change the color to green. There is anther built-in function that we can use for that called *delay*. So we use this in the second instruction of the *loop* function:

```
delay(2000);
```

This causes the Photon/Core to twiddle its virtual fingers for 2000 milliseconds. A millisecond is 1/1000 of a second, so 2000 of them is the same as 2 seconds.

The first two lines of the loop function are repeated two more times, to change the LED color to green and then to blue.

```
void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
```

```
        delay(2000);
    }
```

You might be wondering if you need the final delay command inside loop. Try deleting it and then flashing the code to your Photon/Core again. The updated program will not turn the LED blue at all. In fact, what is happening here is that the LED is being set to blue, but then faster than the human eye, the app is back around to the first line of loop again and the LED will be set to red.

Each time around the loop, the Photon/Core will also do a quick check to see if the Cloud Service is trying to push it a new app. If there is a new app, then your app will be interrupted, and the Photon/Core will take control of the LED back and start flashing purple to indicate that the new app is being “flashed" onto the Photon/Core.

Comments

Project 1 was simply to blink the blue LED next to D7 using the example program provided by Spark. It’s actually a more complex program than that of Project 2, so let’s build on what you have seen in Project 2 and now work through Project 1. Along the way, you will encounter various programming ideas.

It will help to have the code for the example app “Blink an LED” open in the code editor of the Web IDE as the code is examined. Here’s the code. It actually gets pretty wide so the right-hand side of the text has been truncated in the listing below.

```
// Define the pins we're going to call pinMode on
int led = D0;  // You'll need to wire an LED to this one to
see it blink.
int led2 = D7; // This one is the built-in tiny one to the
right of the USB jack

// This routine runs only once upon reset
void setup() {
  // Initialize D0 + D7 pin as output
  // It's important you do this here, inside the setup()
function rather than outside it or in the loop function.
  pinMode(led, OUTPUT);
  pinMode(led2, OUTPUT);
}
```

```
// This routine gets called repeatedly, like once every 5-15
milliseconds.
// Spark firmware interleaves background CPU activity
associated with WiFi + Cloud activity with your code.
// Make sure none of your code delays or blocks for too long
(like more than 5 seconds), or weird things can happen.
void loop() {
  digitalWrite(led, HIGH);   // Turn ON the LED pins
  digitalWrite(led2, HIGH);
  delay(1000);               // Wait for 1000mS = 1 second
  digitalWrite(led, LOW);    // Turn OFF the LED pins
  digitalWrite(led2, LOW);
  delay(1000);               // Wait for 1 second in off mode
}
```

The first thing to notice is that the program has a whole load of lines that start with *//*. Starting a line with *//* indicates that the line is not actually program code at all, but a comment that tells the reader something about the code. The line below tells us pretty much all we need to know about what the next couple of lines are for.

```
// Define the pins we're going to call pinMode on
```

The two lines that follow are a mixture of real code and comment. The *//* word is used again, after the semi-colon to indicate that the rest of the line is also a comment.

```
int led = D0;  // You'll need to wire an LED to this one to
see it blink.
int led2 = D7; // This one is the built-in tiny one to the
right of the USB jack
```

The commenting in this particular app is really quite excessive, and is deliberately very detailed to help the novice try and work out what on earth is going on with the program.

Variables

The two lines above define something called *variables*. Variables are an incredibly useful programming concept. They allow you to give something a meaningful name.

In the case of the first of the lines, there is a pin name D0. That's fine. We know that this refers to one of the physical pins on the

Photon/Core that can be used as an input or output, but it says absolutely nothing about what that pin is to be used for in the rest of the app. Actually, as the comment says, an LED is going to be attached to D0 (you will do this in chapter 4). Later in the code, when we want to turn than pin on and off, its nicer to refer to that pin as the LED pin rather than the actual pin number. The command below is said to assign the value D0 to the variable named *led*.

```
int led = D0;
```

Once variable has been assigned the value D0, from then on, we can refer to pin D0 as *led* rather than D0. The word *int* at the start specifies the *type* of the variable. The type int means a number, and while you might argue that D0 does not look like a number, actually, behind the scenes, D0 is a number used internally be the Photon/Core.

In a later section we will explore the concept of variable types.

Now that a variable called *pin* has been defined to refer to D0, you can use that variable both in the *setup* function, where we set the *pinMode* of the led pin and in the *loop* function where it is turned on and off to make it blink.

Using a variable like this has the great advantage that, if you decided to use pin D3 instead of D0, then you would only have to change the single line

```
int led = D0;
```

to

```
int led = D3;
```

Morse Code

Now, you might wonder why a book about the latest IoT hardware like the Photon would be talking about a 19th century invention called Morse Code. The answer lies in the fact that there are only so many things you can do with an LED, and flashing out Morse code signals is one of the more interesting ones.

Morse code represents letters and numbers as short (dot) or long (dash) beeps or flashes of light. For example the letter *A* is represented as .- (dot dash), one short pulse followed by one long pulse. Although there are a few variants of Morse code, we will use International Morse Code here.

If you were to convert the word PHOTON to Morse code, the resulting message would be: .--. .... --- - --- -. Morse code does not distinguish between upper and lowercase letters.

International Morse Code

The table below lists the International Morse code letters and digits.

A	.-	N	-.	0	-----
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-		
L	.-..	Y	-.--		
M	--	Z	--..		

Flashing SOS

SOS is ... --- ... in Morse code and is used by someone in distress, usually on the high seas and usually a hundred years ago, but lets keep up the pretense that Morse code is still somehow relevant.

We have already made an LED blink, so let's look at how we can apply what we have learnt to flashing a distress signal (albeit, on the small RGB LED).

As a reminder, here is the code from Project 2.

```
void setup() {
    RGB.control(true);
}

void loop() {
    RGB.color(255, 0, 0);
    delay(2000);
    RGB.color(0, 255, 0);
    delay(2000);
    RGB.color(0, 0, 255);
    delay(2000);
}
```

If you typed this code in and got it working in Chapter 2, then all is well and good. If not, then the easiest way to get this into the IDE and then onto your Photon/Core is to click on the Libraries button in the Web IDE and find the library called PHOTON_BOOK. You will find all the project programs used in this book as examples.

Select the file p_02_RGB_LED and then click the button USE THIS EXAMPLE. A copy of the program will open up in the editor and you can then flash it to your Photon/Core. Note that this will be your own copy of the original program, so you can modify it if you wish.

Flash the program onto your Photon/Core, just to remind yourself of the process and see what it does.

While multi-color Morse code could be useful, we should just start by doing some blinking in white light. Change the code of Project 2 to simply blink white (255, 255, 255) as shown below:

```
void setup() {
    RGB.control(true);
}

void loop() {
    RGB.color(255, 255, 255);
    delay(500);
    RGB.color(0, 0, 0);
```

```
        delay(500);
    }
```

You can also find this code in the examples library as ch_03_blink_white.

The sequence of instructions in loop is now:

1. set the RGB LED to white (red, green and blue all maximum brightness)
2. delay half a second
3. turn off the RGB LED
4. delay half a second
5. start again

Flash the app onto your Photon/Core and you should see the LED blink white once per second.

To flash SOS (... --- ...) we can just add some more lines to loop and vary the delays. A dash is three times as long as a delay and the gap between one dot or dash and the next should be the same as a dot. The gap between one letter and the next should be the same as a dash, so change the program so that it appears as below. Note that using copy and paste will save you some time.

```
void setup() {
    RGB.control(true);
}

void loop() {
    // flash S
    // dot
    RGB.color(255, 255, 255);
    delay(200);
    RGB.color(0, 0, 0);
    delay(200);
    // dot
    RGB.color(255, 255, 255);
    delay(200);
    RGB.color(0, 0, 0);
    delay(200);
    // dot
    RGB.color(255, 255, 255);
    delay(200);
    RGB.color(0, 0, 0);
```

```
        delay(600);
        // end of S
        // flash O
        // dash
        RGB.color(255, 255, 255);
        delay(600);
        RGB.color(0, 0, 0);
        delay(200);
        // dash
        RGB.color(255, 255, 255);
        delay(600);
        RGB.color(0, 0, 0);
        delay(200);
        // dash
        RGB.color(255, 255, 255);
        delay(600);
        RGB.color(0, 0, 0);
        delay(600);
        // end of O
        // flash S
        // dot
        RGB.color(255, 255, 255);
        delay(200);
        RGB.color(0, 0, 0);
        delay(200);
        // dot
        RGB.color(255, 255, 255);
        delay(200);
        RGB.color(0, 0, 0);
        delay(200);
        // dot
        RGB.color(255, 255, 255);
        delay(200);
        RGB.color(0, 0, 0);
        delay(600);
        // end of S

        delay(2000); // delay before repeat
    }
```

While this code works just fine, there is much more of it than there needs to be, as a lot of the code is repeated. In the next section you will use functions to shorten the code considerably.

Functions

We have been happily using some of the functions such as *delay* and *RGB.color* that are built in to the Spark.io system. You can also write your own functions to group together lines of code that you might want to use over and over again in your program.

For example, wouldn't it be great if there was a function called *flash*, by which we could just pass as parameters the duration of the flash and how long to pause before any further flashing. Well, you can create such a function and then modify the SOS program to use it. The function looks like this:

```
void flash(int duration, int gap) {
    RGB.color(255, 255, 255);
    delay(duration);
    RGB.color(0, 0, 0);
    delay(gap);
}
```

The function starts with the word *void*, because this function does not return any value. Next we have the name of the function, and I have decided to call it *flash*. Inside the parentheses, two parameters are defined, *duration* and *gap*. Both need to have the type specified for them. Since they are a whole number of milliseconds the type *int* should be used.

The body of the function is just like a section of the long-winded code we had before, but now, instead of having fixed numbers like 200 or 600 inside the delay functions, we have what look like variable names of duration and gap. These values will be passed into the function when we use it in *loop*.

To see how this works, lets now look at the whole program, which you can find in ch_03_SOS_function.

```
void setup() {
    RGB.control(true);
}

void loop() {
    // flash S
    flash(200, 200);
    flash(200, 200);
    flash(200, 600);
    // flash O
    flash(600, 200);
    flash(600, 200);
    flash(600, 600);
    // flash S
    flash(200, 200);
    flash(200, 200);
    flash(200, 600);

    delay(2000); // delay before repeat
}

void flash(int duration, int gap) {
    RGB.color(255, 255, 255);
    delay(duration);
    RGB.color(0, 0, 0);
    delay(gap);
}
```

This is great, because the this program is just 27 lines long rather than the 59 lines of the previous version. Smaller programs are better, being easy to maintain. Programmers have an acronym that they use - DRY - “Don’t Repeat Yourself” and will talk of good code being DRY.

Now, to flash the letter S, you can just call *flash* three times in a row. On the last occasion, longer delay is used to allow a gap between letters.

We could take this program a step further and create two new functions called *flashS* and *flashO* that call *flash*. This would simplify the code in loop even more. The result of this can be found in the file ch_03_SOS_function2.

```
void setup() {
    RGB.control(true);
}

void loop() {
    flashS();
    flashO();
    flashS();

    delay(2000); // delay before repeat
}

void flash(int duration, int gap) {
        RGB.color(255, 255, 255);
        delay(duration);
        RGB.color(0, 0, 0);
        delay(gap);
    }

    void flashS() {
        flash(200, 200);
        flash(200, 200);
        flash(200, 600);
    }

    void flashO() {
        flash(600, 200);
        flash(600, 200);
        flash(600, 600);
    }
```

This hasn't made the program any shorter, but an interesting thing has happened. Some of the comments in loop have become so obvious that they can now be removed. For example, there is little point in preceding the line:

```
flashS();
```

with a comment like:

```
// flash S
```

Generally, the fewer comments that are needed to "explain" the code the better. The code has become less obscure and is starting to be self-explanatory.

Incidentally, it does not matter where in the program you write the code for a function. However, the convention is to have the *setup* and *loop* functions at the top of the file, as these are the root of any later calls to other functions. In other words, *setup* and *loop* will always be the starting point for anyone reading through the program to work out what it does.

Types

The SOS program is now looking pretty neat. You can start at *loop*, and understand that *loop* is going to call *flashS* and *flashO*. You can then look at *flashS* and see that *flashS* calls *flash*. Its all quite easy to "read". The program does, however, have one weakness: should you want to make it flash SOS faster or slower, you would have to go through and change all occurrences of 200 and 600 to something else. Its easy to fix this so that you only have to change one variable.

The modified version can be found in the file ch_03_sos_vars.

```
int dot = 200;
int dash = dot * 3;

void setup() {
    RGB.control(true);
}

void loop() {
    flashS();
    flashO();
    flashS();

    delay(2000); // delay before repeat
}

void flash(int duration, int gap) {
    RGB.color(255, 255, 255);
    delay(duration);
    RGB.color(0, 0, 0);
    delay(gap);
}

void flashS() {
    flash(dot, dot);
    flash(dot, dot);
    flash(dot, dash);
}

void flashO() {
    flash(dash, dot);
    flash(dash, dot);
    flash(dash, dash);
}
```

At the top of the file, two new variables, *dot* and *dash* have been added. The variable *dot* is set to 100. This will be the duration of the delay for a dot flash. The variable *dash* is the duration of a dash and we could just set this directly to 600, but then if we wanted to make the SOS blinking faster or slower, we would have to change both variables. Since a dash is always three times as long as a dot, we can express that in our code like this:

```
    int dash = dot * 3;
```

Here, you are defining a new variable called *dash* (of type *int*) and then assigning it a value of dot * 3. In the C language * means multiply.

As well as * for multiply, you can use + for add, - for subtract and / for divide.

Change *dot* to 100 and then flash it onto your Photon/Core. Notice how the blinking is now faster.

The int Type

We have established that the type *int* is used for whole number numbers. These numbers can be positive or negative or 0 but must be in the range -2,147,483,648 to +2,147,483,647.

Note that if you have come from the world of Arduino where the range of numbers in an *int* is much smaller, this will be a pleasant surprise for you. Although Spark C has other integer types like long (same range as int) as well as signed and unsigned versions of the types, there is little point in ever using anything other than *int* to represent whole numbers.

The float Type

Variables of type *int* are great for whole numbers, like pin numbers or counting, but, occasionally, you might want to use numbers with a decimal place in them. For example, if you are reading temperatures from a sensor, the nearest degree might not be precise enough and you might want to represent a value of say 68.3 degrees.

For these kind of numbers you cannot use *int.* You use a type called *float*. They are called *floats*, because they are "floating point" numbers. That is, the position of the decimal place could be anywhere in the number; they are not say fixed to two decimal places.

Here is an example of a float:

```
float temperature = 68.3;
```

Floats have a vast range, from -3.4028235E+38 to 3.4028235E +38. The notation E+38 means with 38 zeros on the end of the number. These numbers are as big as you could ever need. For the mathematically minded, floats here are 32 bits.

So, if floats have a much wider range, then why bother with restricted int values? Surely we could just use floats, and if they need to be whole numbers, just have .0 on the end.

The reason this is not done is that floats are deceptive. Although they have a wide range of possible values they do this by using an internal representation that effectively approximates the number. Although they represent numbers that can be 39 digits long, if you performed some arithmetic on two numbers that were that long, the result would not be exact and under some circumstances, you can get results equivalent to 2 + 2 = 3.9999999999999999999.

The likelihood of such errors can be reduced by using the type *double* which uses a 64 bit representation and is, therefore, a lot more precise.

Arithmetic in *floats* is also slower than with ints, and, if you use *doubles*, it takes up twice as much memory. So, avoid using *floats* or *doubles* unless you have a good reason for it, such as the temperature example.

Other Types

As this book progresses, you will meet other data types such as *boolean,* which represents the values *true* and *false,* as well as more complex types, like *strings*, which are used to represent text.

Arrays

The variables used so far have all been just single values. Sometimes you need to use a data structure that represents a list of values. For example a list of flash delays.

Open the app ch_03_SOS_Function in the Web IDE. This is taking a step backwards, but this version is quite suitable to modify to use an array of delay values.

The loop function of this program looks like this:

```
void loop() {
    // flash S
    flash(200, 200);
    flash(200, 200);
    flash(200, 600);
    // flash O
    flash(600, 200);
    flash(600, 200);
    flash(600, 600);
    // flash S
    flash(200, 200);
    flash(200, 200);
    flash(200, 600);

    delay(2000); // delay before repeat
}
```

Looking at this code, if you removed the comments, you would have nine flash commands one after the other, each with a series of values both for duration and gap. We found one way to simplify this code using functions called *flashS* and *flashO*, but another way to simplify the code would be to use two arrays: one for the duration and one for the gap. We could then go through each position of these arrays in turn, calling *flash* with the values. That way, we could change the Morse Code message to anything we liked, just by changing the contents of the arrays.

This is how you define an array of *ints*:

```
int durations[] = {200, 200, 200, 600, 600, 600, 200, 200,
200};
```

Notice that the variable name now has [] after it to indicate that it is an array rather than a single value. The values separated by commas inside { and } are the values to be put into the array. In this case, that is each of the durations used in loop.

We could do the same for the gap parameters of the calls to flash like this:

```
int gaps[] = {200, 200, 600, 200, 200, 600, 200, 200, 600};
```

It's almost like there are two tapes with the same numbers on them that are then going to be fed through a machine that will use one number for the flash duration and the other for the gap before the next position on the tape.

Loops

Now that we have the two arrays, we need a way to step over each of those values in turn and call *flash* using them.

This type of thing is called 'looping' or 'iteration' and there is a useful language command in C for doing this called *for*.

Here is the modified program that uses *for* to loop over each element of the arrays and flash them. If you want to try out the completed program its in ch_03_SOS_Array.

```
int durations[] = {200, 200, 200, 600, 600, 600, 200, 200,
200};
int gaps[] =      {200, 200, 600, 200, 200, 600, 200, 200,
600};

void setup() {
    RGB.control(true);
}

void loop() {
    for (int i = 0; i < 9; i++) {
        flash(durations[i], gaps[i]);
    }

    delay(2000); // delay before repeat
}

void flash(int duration, int gap) {
    RGB.color(255, 255, 255);
    delay(duration);
    RGB.color(0, 0, 0);
```

```
        delay(gap);
    }
```

The syntax of the *for* command highlighted above is confusing. It looks a bit like a function call, but it isn't because it has parts separated by semicolons - surely these should be commas?

If you are new to programming, you are probably best off to just use the *for* loop for counting, copy the line starting with *for* and change 9 to whatever number you want *i* to count up to. Its not you; the syntax really is a bit screwy.

What is actually going on is that the first thing inside the parentheses of the *for* command defines a variable. In this case the variable is called *i* (for index) and this variable is initialized to 0 and will be used to keep track of our position in the arrays.

Then there is a semicolon and the expression i < 9. This is the condition for staying in the loop. In other words, the program will not escape from *for* and go on to do the last *delay* command while *i* is less than 9. This means that our program will hang unless we do something to change the value of *i*. That is where the last part of the stuff inside the () of the *for* command come into effect. The expression *i++* means add 1 to the value of *i*.

So, each time around the *for* loop, flash will be called with the i th value of durations (thats what [i] means) and the i th value of gaps. The variable *i* will then have 1 aded to it and then for will call *flash* again but this time *i* will have changed from 0 to 1. This will continue until *i* gets to 9 and then because *i* is now greater than 8 the *for* loop is finished and the program will move on to the last line of the *loop* function and delay for two seconds.

As we mentioned earlier, if we change what is in the arrays, we can change the message completely. To try this, load up the example ch_03_flash_photon. Note that the maximum value of *i* in the for loop has been changed from 9 to 16 because the arrays have got bigger.

You may have noticed that *i* starts at 0 rather than 1. In C, the first element of an array is the zeroth element rather than the first element.

Strings

The Morse Code example is gradually evolving away from something that very specifically can only flash SOS into something that will eventually be able to flash any message that we tell it to.

To do this, we need a way for the program to represent text and for this we will introduce a new type called String. Here is how you define a String variable.

```
String message = "My Photon speaks Morse";
```

The value given to the string is enclosed in double quotes. If you are used to languages like Python that will allow you to use double or single quotes, then beware because only double quotes will do for C. C does use single quotes but these are reserved for single characters of type *char*. You will meet the *char* type a little later on.

There are various things that you can do with strings. For a start, you can find out how many characters they contain like this:

```
String message = "My Photon speaks Morse";
int len = message.length();
```

The function *length* is a special kind of function called a *method*. The difference between a method and a function is that a method is owned by a type, so here strings have a method called *length* that you can call using the dot notation after the string variable name. In this case, the *length* method returns a value (the number of characters in the string) that can then be assigned to an *int* variable.

As we mentioned earlier, C has a type specifically for single characters. Since a string is made up of a sequence of characters, its a bit like an array. You can access any of the strings individual characters using the method *charAt()*. The following example will assign the value “P" to the variable letter.

```
String message = "My Photon speaks Morse";
char letter = message.charAt(3);
```

Just like arrays, the index positions start at 0 rather than 1.

There are lots of other things that you can do with strings, we will encounter these as we finish the Morse Code Flasher project.

Ifs

The C *if* command is one that you will find in pretty much any program of any size. It allows the program to make decisions as to what to do based on some condition. It works much the same way as the English word *if* in a sentence. So, the English statement “if the character is a space, delay for 7 dots worth of time" (the gap between words in Morse Code is 7 times the length of a dot) would be written in code as:

```
if (letter == ' ') {
   delay(dot * 7);
}
```

Notice how, when comparing letter to a space character, the two equals signs are used. This is to distinguish it from the single equals sign used to give a variable a value. A common source of programs not doing what was expected is that an = was used instead of an ==.

You can use == to test if two things are equal, or != to test that they are not equal. You can also use < (less than), <= (less than or equal to), > (greater than) or >= (greater than or equal to).

It is also possible to combine more than one condition using && (and) and || (or). For example the block of code inside this if statement will only be run if the value of letter is between ‘A’ and ‘Z’. You can use these comparisons on ints, floats and chars, but not Strings.

```
if (letter >= 'a' && letter <= 'z') {
   // its an uppercase letter do something!
}
```

The *else* command works with *if*. The usage is the same as in English. For example in English you could write: “if the temperature is less than 70 turn on the heating else turn it off.”

In C, this might look like this:

```
if (temperature < 70.0) {
    heatingOn();
else {
    heatingOff();
}
```

I have assumed that two C functions called *heatingOn* and *heatingOff* exist to actually control the heating.

Project 3. Morse Code Flasher

In this project, you will put together everything you have learned about programming the Photon/Core. This project will flash any plain text held in a string as Morse Code.

In Project 4 you will use this same code with an external LED and in Project 8 flash Morse Code messages using a web interface.

Software

You can find the code for this project in the example library as p_03_Morse_Flasher. I suggest you open this file up in the Web IDE while you are looking at the code.

The first line of code specifies the message that is to be flashed:

```
String message = "My Photon speaks Morse";
```

You can change this message to anything you like.

Following this are the variables *dot* and *dash* that set the timings for the flashes. Then we have an array of strings that defines all the letters in Morse code:

```
int dot = 200;
int dash = dot * 3;

String letters[] = {
  ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....",
"..",    // A-I
  ".---", "-.-", ".-..", "--", "-.", "---", ".--.", "--.-",
".-.",  // J-R
  "...", "-", "..-", "...-", ".--", "-..-", "-.--", "--.."
     // S-Z
};
```

Each of the letters A to Z is represented by one element of the array. The first element of the array is the string ".-" for A, the second "-..." for B etc. This array will be used to look up the series of dots and dashes that must be flashed in order to flash a particular letter as Morse Code.

The *setup* function just takes control of the RGB LED and the *loop* function now just contains the function call to *flashMessage*.

```
void setup() {
    RGB.control(true);
}

void loop() {
    flashMessage(message);
    delay(5000); // delay before repeat
}
```

This type of programming is called “programming by intention." You know what you intend to do in the *loop* function: you want to flash the message and then delay for a while before flashing the whole message again. You can worry about how the *flashMessage* function will actually do its job later.

In fact, implementing *flashMessage* is the next step. Logically what you need to do, to be able to flash a whole message is to flash each letter of that message in turn.

```
void flashMessage(String message) {
    for (int i = 0; i < message.length(); i++) {
        char letter = message.charAt(i);
        flashLetter(letter);
    }
}
```

So, all flashMessage needs to do is to step over each letter in turn and call a function *flashLetter* on that letter. Once again, we are deferring the actual LED flashing work to another function that we have not written yet called *flashLetter*. See how nicely the problem is breaking down for us?

```
void flashLetter(char letter)
{
    if (letter >= 'a' && letter <= 'z') {
        flashDotsAndDashes(letters[letter - 'a']);
    }
    else if (letter >= 'A' && letter <= 'Z') {
        flashDotsAndDashes(letters[letter - 'A']);
    }
    else if (letter == ' ') {
        delay(dot * 7); // gap between words
    }
}
```

The *flashLetter* function is a little more complicated. First we need to use *if* commands to decide if the letter is an uppercase letter or a lowercase letter, or a space character. If its a lowercase letter, then we find the string containing the dots and dashes for that letter using the letters array and supplying the character code minus the code for ‘a’ to get the right sequence of dots and dashes from the *letters* array.

If the letter is uppercase, then you need to subtract the character code for “A” rather than “a.”

Finally if the letter is the space character, that indicates that it’s the end of a word and the Morse code standard is to leave a gap of seven dots worth of time between each word.

Yet again, we are deferring the job of making the flashes to a function called *flashDotAndDashes*.

```
void flashDotsAndDashes(String dotsAndDashes) {
    for (int i = 0; i < dotsAndDashes.length(); i++) {
        char dotOrDash = dotsAndDashes.charAt(i);
        if (dotOrDash == '.') {
            flash(dot);
        }
        else {
            flash(dash);
        }
        delay(dot); // gap between dots and dashes of a letter
    }
    delay(dash - dot); // gap between letters of a word
}
```

The parameter passed to *flashDotAndDashes* is a String. This string will be made up of the dots and dashes for a particular letter. For example -... for B. So to flash out those dots and dashes, you need to step over each character in that string, flashing the dot or dash.

After you have flashed each dot or dash, you need one dots worth of delay before you start on the next dot or dash and when you have completed the whole sequence you need another delay of duration a dash. However, you will already have had a delay of one dot after the last dot or dash was flashed, so the delay is for the duration of a dash minus a dot.

And still we are not actually controlling the LED. This happens in the flash function, which just takes the duration of the flash as a parameter. It is this function that actually turns the LED on and off.

```
void flash(int duration) {
    RGB.color(255, 255, 255);
    delay(duration);
    RGB.color(0, 0, 0);
}
```

That is all there is to this project. Try changing the message and also the speed of the Morse Code by altering the value of the *dot* variable.

Summary

This chapter has only really touched on programming the Photon/Core. Enough to get you started. As the book progresses you will learn more about programming the Photon.

In the next chapter, you will learn how to connect hardware to the Photon/Core using solderless breadboard.

4/Breadboard

In this chapter we will deal with attaching some simple external electronics to the Photon or Core, including LEDs and switches. To connect electronic components like LEDs to the Photon/Core you will use something called solderless breadboard.

Solderless breadboard (or just breadboard) was invented as a tool for electronics engineers to prototype their designs before committing the projects to a more permanent soldered form. They do, however, make an excellent tool for you to experiment with electronics and make your own projects without the need for any soldering.

Figure 4-1 shows a breadboard with a Spark device, an LED and a resistor attached to it.

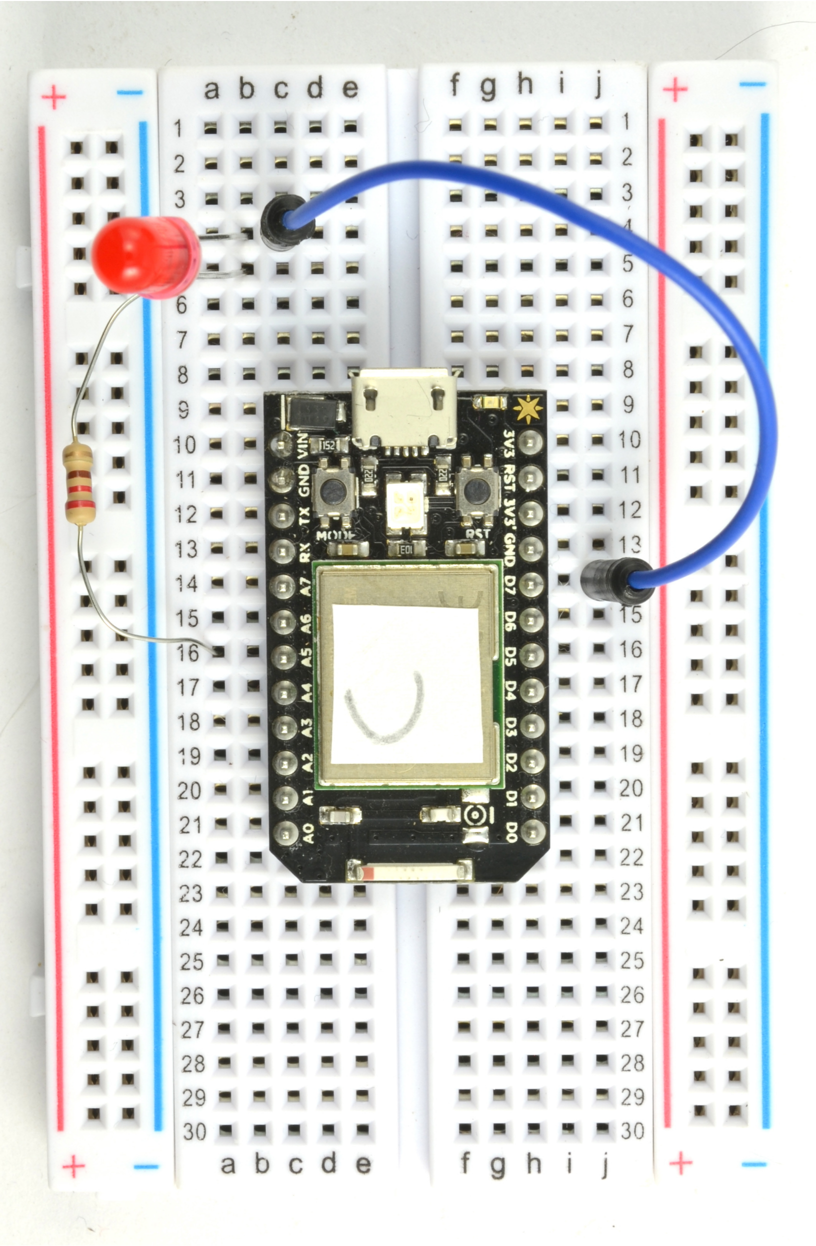

a b c d e
f g h i j
VIN
GND
TX
RX
A7
A6
A5
A4
A3
A2
A1
A0
3V3
RST
3V3*
GND
D7
D6
D5
D4
D3
D2
D1
D0
MODE
RST

Figure 4-1. *Figure 4-1. A Core on Breadboard*

When you buy a Photon or Core, one of the options available to you is for it to include a breadboard and header pins. If you did not select this option, but rather bought a bare Photon, then don’t worry, you will find sources for the header pins and breadboard that you will need in Appendix A.

How Breadboard Works

Behind the holes in the front plastic face of a breadboard there are metal clips designed to grip wires and component leads. These clips are arranged into rows of five in two banks of 30 rows. At least they are in the fairly standard “half-sized" breadboard that is included as an option when you buy a Photon.

Down each side of the breadboard are pairs of long clips that are marked with red and blue lines on the front of the breadboard. These are often used when connecting power to various parts of your breadboard design.

Figure 4-2 shows a breadboard that has been taken apart and one of the clips removed, so that you can see how it works behind the plastic.

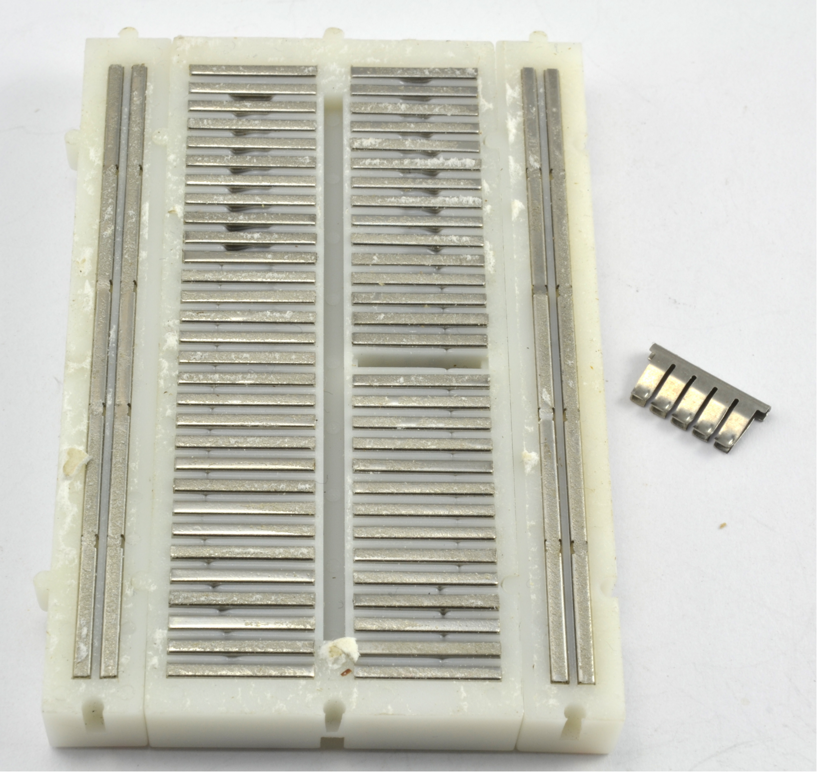


Figure 4-2. *Figure 4-2. A Breadboard Disassembled*

Attaching an LED

LEDs emit light when a current passes through them. Unfortunately, LEDs are kind of greedy for current and have so little restraint that, when fed with a source of voltage like an output pin on a Photon/Core, they will draw as much current as they can. This can lead to their overheating, shortening their life, or even damaging the Photon.

To prevent such problems, LEDs are used with the electronic equivalent of a gastric-band that restricts the flow of current into the LED. The component that restricts the flow of current in this way is called a resistor.

Figure 4-3 shows a schematic diagram for using a resistor with an LED.

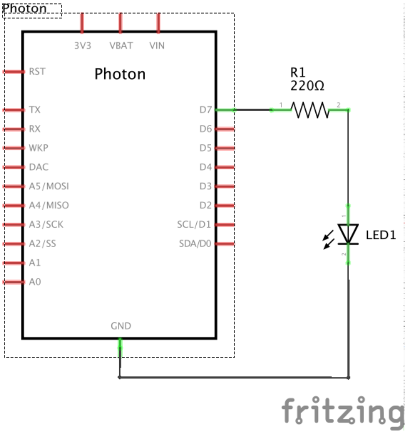


Figure 4-3. *Figure 4-3. Limiting the Current to an LED with a Resistor*

The two pins of the Photon that are used are GND and D7. GND or ground is the base voltage against which any other voltages on the Photon are measured. For example the 3V3 pin is 3.3V above (more positive than) GND. Current will only flow from a higher voltage to a lower one, so if the digital output D7 is at 0V (called LOW) no current will flow, but if it is at 3.3V (HIGH) then the current can flow out of D7 through the resistor (zig-zag line) through the LED and down to GND. You may find that it helps to think of current flowing like water from somewhere high down to sea-level.

Digital Outputs

The pins on your Photon/Core labelled D0 to D7 as well as the pins A0 to A5 can all be used as digital outputs. That is, you can write some commands in your program that will turn them on or off. More accurately the pins are either set to 3.3V or 0V (GND).

Because the pins can be used as either inputs or outputs, the first thing you need to do in your setup function, for all the pins you are using, is to specify whether that pin is to be used as an input or an output. In the Digital Inputs section in this chapter, you will find out how to specify a pin as being an input. To specify that a pin is going to be used as an output, include the following command in your setup function:

```
pinMode(pin, OUTPUT);
```

When it comes to actually setting the pin HIGH or LOW, you use the command *digitalWrite*. So, assuming that the variable pin is set to one of the pins (say D7), then to set the pin HIGH you would use:

```
digitalWrite(pin, HIGH);
```

and to set it low, you would use:

```
digitalWrite(pin, LOW);
```

In Project 4, you will use these commands to make an external LED blink Morse Code.

Project 4. Morse Flasher (External LED)

Now that you know how to use an LED with a Photon or Core, you can change the Morse Code flasher of Project 3 to use an external LED rather than the built-in RGB LED.

Parts

To build this project, you are going to need the following parts in addition to your Photon or Core.

Table 4-1. *Parts Bin*

Part	Description	Appendix A code
LED1	Red or for that matter any other color LED	C2
R1	220Ω resistor	C1
	Male to Male jumper wire	H4
	Half-sized breadboard	H5

These components may be included in the Spark Maker's Kit.

Hardware

We can convert the schematic of Figure 4-2 into a breadboard layout as shown in Figure 4-3.

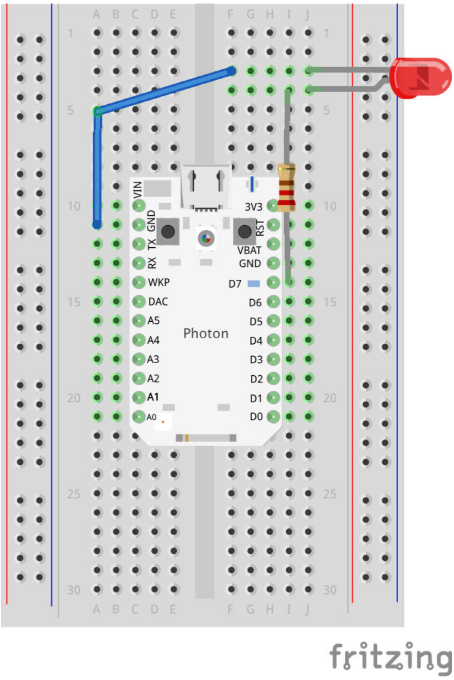

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 4-4. *Figure 4-4. Breadboard Layout for LED and Photon*

Start by pushing one end of the resistor (it does not matter which) into the same row as D7 on the Photon/Core and then put the other leg of the resistor into row 4 of the breadboard.

The LED will have one leg longer than the other. The longer leg is the positive leg and this is the leg that should go to the same row as the resistor.

Finally, connect a jumper wire from the row of the breadboard on the lefthand side, which is connected to one of the Photon/Core GND pins, to the top row that the LED is connected to.

Software

If you just want to check that your hardware is working, then you can flash the "Blink an LED" example app from the Examples section of the Web IDE onto your Photon/Core. This is the same program that you ran in Project 1.

When this is running, you should see both the built-in LED and the LED on the breadboard blinking in time.

If it doesn't work, then check over the connections carefully and make sure that the LED is the right way around. Note that if you put the LED in the wrong way around it will not do any harm, it just won't work.

Once you have the basic blink working, you can load up the app from this project, which you can find with all the projects for this book in the public Library called PHOTON_BOOK. The files is called P_04_Morse_External_LED.

There are just a few differences between this program and that of Project 3. They are all concerned with the change from using the built-in RGB LED and pin D7. The first change is the addition of a new variable to specify which pin to use:

```
int ledPin = D7;
```

The *setup* function is also slightly different, because now you need to set the *ledPin* to be an output.

```
void setup() {
    pinMode(ledPin, OUTPUT);
}
```

The only other code comes right at the end of the program in the *flash* function:

```
void flash(int duration) {
    digitalWrite(ledPin, HIGH);
    delay(duration);
    digitalWrite(ledPin, LOW);
}
```

Now, the *flash* function will turn the *ledPin* on and off rather than the built-in RGB LED.

Note that both the built-in tiny blue LED and the LED on the breadboard will both flash the message. The internal LED on pin D7 consumes very little current and has negligible effect on the use of that pin as an output.

Attaching a Switch

Whereas an LED is probably the most common use of a digital output on a microcontroller like the Photon, a switch is probably the most common input.

Figure 4-5 shows the schematic diagram for attaching a switch to digital pin D3 of a Photon.

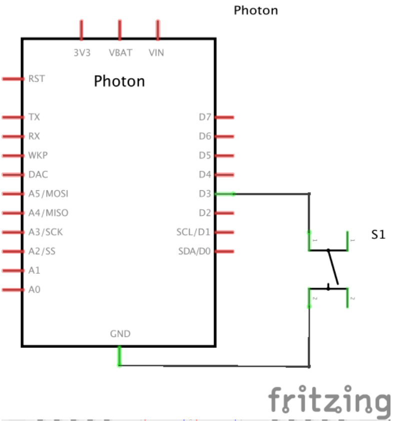


Figure 4-5. *Figure 4-5. Connecting a Switch to a Photon*

The symbol for the switch is a little strange. You can see the switch contacts in the center of the switch and the two left-most terminals of the switch are used. One terminal going to the pin that will act as a digital input and one to GND. So when the switch is pressed, the digital input D3 will be connected to GND (0V).

The type of switch shown here, and used in the next project, is called a tactile push switch and usually has four pins. This gives it greater mechanical strength when its soldered onto a PCB (or used with breadboard). The two unused connections of the switch on the right, just represent these extra pins.

In some schematic diagrams for using switches with a micro-controller, you may find a “pull-up” resistor connected between the digital input and 3.3V. The point of this resistor is to pull the input up to 3.3V (HIGH) until the switch is pressed where the switch will override the pulling effect of the resistor and make the input LOW. This is not necessary with the Photon or Core as these devices digital pins have built-in pull-up resistors that can be enabled in your code.

Switches and Outputs

The switch connects D3 to GND, which is just fine as long as D3 is set to be an input. However, if the program currently running on the Photon/Core is a digital output and that output happened to be HIGH (3.3V) then pressing the switch would connect the 3.3V output to GND.

This is very bad and would quite likely damage the Photon or Core. So when using a Photon with digital inputs, it is a good idea to load the app up first, before you start wiring up the hardware, so that there is no danger of a digital output being short-circuited to GND.

Digital Inputs

Before you can use a pin of the Photon/Core, you have to use the *pinMode* function to set its mode to be an input. Specifying that a pin should be used as an input actually has three possible types of digital input:

- INPUT - no pull-up resistors. This would typically be used for connecting the input to digital outputs of a chip or module
- INPUT_PULLUP - pull-up resistor enabled. Use this when connecting the digital input to a switch that will switch the input to GND
- INPUT_PULLDOWN - pull-down resistor enabled. Use this when connecting the digital input to a switch that will switch the input to 3.3V

While it might seem more logical to use the last option and connect your switches to 3.3V so that pressing the button takes the input HIGH. It is more common to use the INPUT_PULLUP option and connect the switch to GND. This is partly because the Arduino boards do not have the third option, but also if a pin is accidentally left as an output (see warning box) then it will by default be LOW, so an accidental short circuit as a result of pressing the button is reduced.

So, to use the input connected to a switch, as I recommend and as shown in Figure 4-5, you would add the following command to your *setup* function.

```
pinMode(switchPin, INPUT_PULLUP);
```

In project 5, the Morse Flasher project will be modified to only start flashing when a button is pressed.

Project 5. Morse Flasher with Switch

In this project Project 4 will be expanded to include a tactile push switch. When the switch is pressed, the message will be flashed out in Morse Code.

Parts

To build this project, you are going to need the following parts in addition to your Photon or Core.

Table 4-2. *Parts Bin*

Part	Description	Appendix A code
LED1	Red or for that matter any other color LED.	C2
R1	220Ω resistor	C1
S1	Tactile push switch	C3
	Male to Male jumper wires	H4
	Half-sized breadboard	H5

These components are all included in the Inventors Kit and apart from the switch are the same as for Project 4.

Software

You can find the software for this project in the file P_05_Morse_Switch. Get this loaded onto your Photon/Core, before you wire up the switch.

The key differences between this program and the program for Project 4 are that there needs to be a new variable to specify the input pin:

```
int switchPin = D3;
```

In the *setup* function, this new pin needs to be specified as an input with pull-up resistor enabled.

```
void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(switchPin, INPUT_PULLUP);
}
```

Finally, the *loop* function needs to be changed so that the message is only flashed if the button is pressed. We can also do away with the delay on the end of loop, because the message will only be flashed when the button is pressed, rather than every time around the loop.

```
void loop() {
    if (digitalRead(switchPin) == LOW) {
        flashMessage(message);
    }
}
```

The if command tests to see if the result of doing a digital read on the switch pin is LOW. Remember that the input will be LOW when the switch is pressed, otherwise it will be “pulled-up" HIGH.

Hardware

The breadboard layout for this project starts the same as that of Project 4. So, if you still have Project 4 on breadboard, you can just add the switch and two jumper wires that lead to it (Figure 4-6).

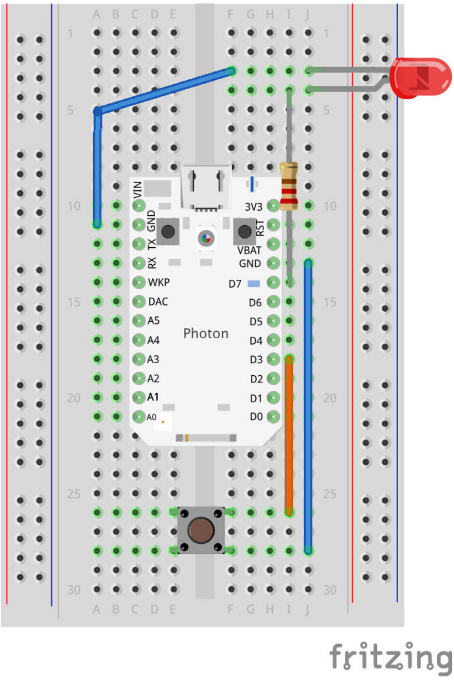

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 4-6. *Figure 4-6. Breadboard Layout for Project 5*

Note that the push switch will just fit across the central divide between the two banks of breadboard rows. This will also ensure that it is the right way around and that the connections will go to a pair of pins that are separated by just one breadboard hole.

Running the Project

Since you have already made all the wiring, you should find that pressing the switch will start the Morse Code flashing.

Analog Outputs

Although you have experimented with digital outputs, that is, turning something on and off, you may also want to say control the brightness of an LED or the speed of a motor. To do this, you need to use an analog output.

Certain of the Photon's pins (pins D0, D1, D2, D3, A4, A5) and the Core's pins (D0, D1, A0, A1, A4, A5, A6, A7) can be used as analog outputs. These outputs are not true analog outputs, but rather they are outputs that control the power to things that they are connected to by rapidly switching on and off (see the side bar on Pulse Width Modulation).

Pulse Width Modulation (PWM)

A PWM output of a Photon or Core will produce 500 pulses every second. Figure 4-7 shows what these pulses might look like.

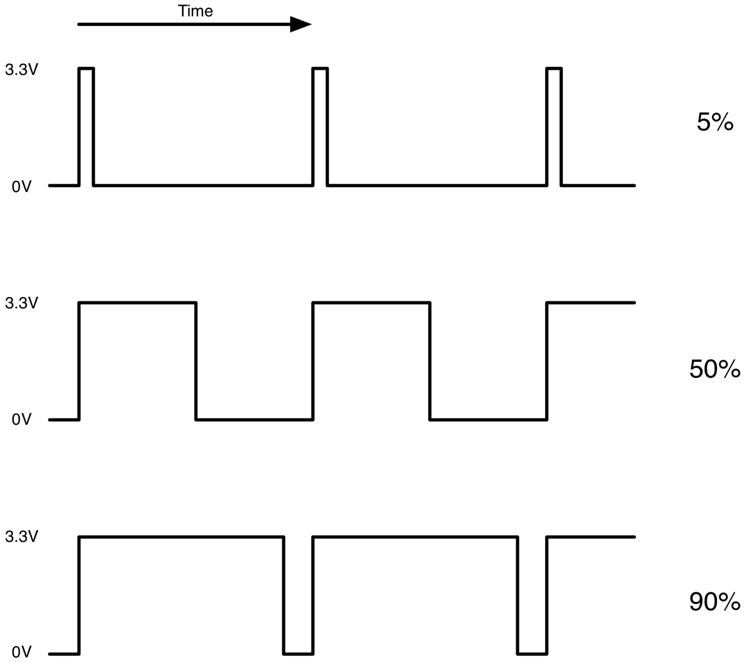


Figure 4-7. *Figure 4-7. Pulse Width Modulation.*

If the pulse is quite short (say only HIGH for 5% of the time), then a relatively small amount of energy to whatever the pin is powering, so if it is an LED, the LED will appear dim, because is actually only on for 5% of the time. It switches so quickly, that the human eye cannot tell that its actually flashing really fast and just registers it as being dim. The same principal can be used to control a motor, only a small “kick" of energy is being given to the motor with each pulse.

If the PWM output is high for half of the time, then whatever is being powered (the load) will receive half-power and if the PWM output is high for most of the time then it will get a lot more power.

analogWrite

To control a PWM output, the analogWrite command is used. This command takes two parameters, as illustrated in the example below:

```
analogWrite(ledPin, 127);
```

The first parameter is the pin to be used as a PWM output and the second parameter is called the duty cycle. That is how long each pulse should be high. The value of duty cycle is a number between 0, which means permanently off, and 255 which means permanently on. So a value of 127 will put the output at pretty much half power.

If you are using a pin as a PWM output, you still have to define it as an output using pinMode.

An Example

As an example of using PWM, you can modify the hardware that you used in Project 5 to use a pin that is PWM capable. To do this, just move the resistor lead currently connected to D7 to A5. The revised breadboard layout is shown in Figure 4-8.

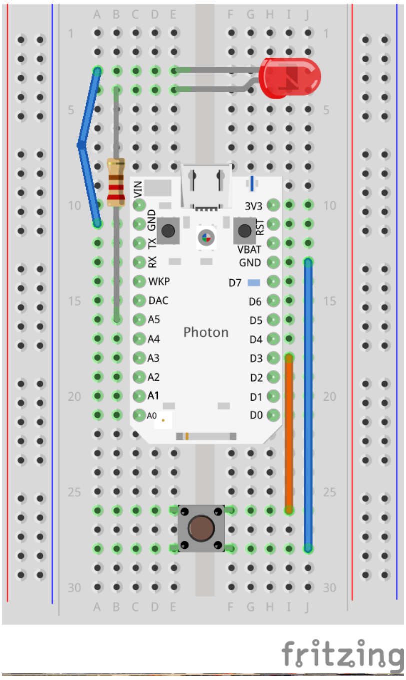

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 4-8. *Figure 4-8. Revised Breadboard Layout for PWM*

Flash the following program that can be found in the PHOTON_BOOK examples project as ch_03_pwm_output.

```
print('hello world')
```

Initially, the LED will be off, but then each time that you press the button it will get a little brighter. After a few button presses the LED will cycle back around to being off.

A variable called brightness is used to keep track of the current duty cycle of the ledPin. This is initialized to 0.

The loop function starts with an if command that checks to see if the button is pressed. If it is then 25 is added to the variable brightness. If the value of brightness exceeds 255 then it is set back to 0.

The duty cycle of the ledPin is set to the current value of brightness and then a delay of 200 milliseconds prevents the value of brightness immediately changing again because the button is still pressed.

A Real Analog Output

The Photon has a “true” analog output that is not present on the older Core. This pin is immediately above A5 and is labelled DAC (Digital to Analog Converter). Unlike the pins that are capable of PWM, the DAC pin does not use pulses but does actually change the voltage on it output between 0 and 3.3V.

To set the voltage of the DAC pin you use analogWrite just like you would with a PWM pin, for example:

```
    analogWrite(DAC, 127);
```

Summary

In this chapter, you have learned the basics of digital inputs and outputs. As yet, we have not touched on analog outputs and inputs, but you will meet these in later chapters as the book progresses.

In Chapter 5, you will learn about the Internet of Things and how to control electronics over the Internet.

5/The Internet of Things

Aside from actually programming the Photon/Core over the Internet, everything we have done up to now with the these could have been done with a device such as an Arduino without any internet connection. While the Photon/Core is actually pretty good in the role of an Arduino replacement, to not use its Internet of Things capabilities is to miss out on its most exciting feature.

In this chapter, you will learn how to communicate with a Photon or Core over the Internet to both issue commands to the device and also take readings from its sensors.

Spark Functions

Many IoT devices require you to do some pretty hairy network programming to have your device communicate with the Internet. Not so the Photon and Core. Spark have managed to provide a simple and easy to use framework for using your Photon/Core over the Internet.

In a later section of this chapter, you will learn about reading information from the Photon/Core. In this section I will concentrate on sending instructions to the Photon over the Internet. For example in the next project you will be able to send an HTTP post to a URL that then turns on an LED on the Photon.

The key to telling your Photon or Core to do things is the concept of a “Function." To distinguish Spark Functions from C language functions in your programs, I will use an uppercase **F** when referring to Spark **F**unctions.

A Function associates a command that you can send to the Photon/Core with a C function on that device’s app. Whenever the device receives that command, it runs the associated C function. Rather like a regular C function, the function in the app also receives a parameter. The parameter is always of type String, and it's up to your function in the program as to how it uses that parameter, if indeed it uses the parameter at all.

Actions and Functions

Note that in earlier versions of the Spark software, Functions used to be called *actions*. This change was made to tie in better with the naming conventions of IFTTT (see Chapter 6).

You can define up to four Functions in the same program. This may sound restrictive, but remember, you can pass data to the function in its parameter and therefore do different things depending on that data.

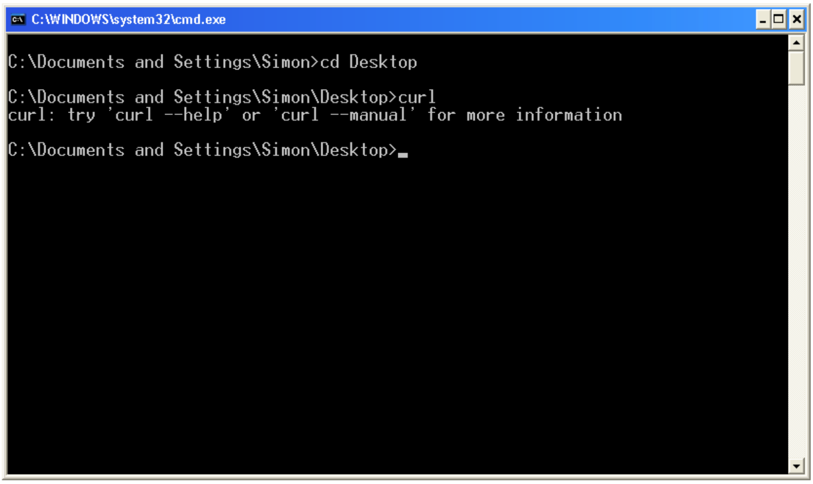
In Project 6 you will make a simple example that will allow you to turn the built-in LED on pin D7 of the Photon/Core on and off from a command line. But first, if you are a Windows user, you will need to set up your environment.

Sending actions to a Photon/Core requires you to send HTTP post requests to the Spark service. A handy way to do this is using the command line tool called *curl*. If you are a Mac or Linux user, then the good news is that 'curl' will already be installed and ready to use. If you are running Windows, then you will need to install curl as described below.

Open your web browser on http://curl.haxx.se/download.html and scroll down to find the Win section of the downloads. Unless you have a really old computer, you will probably need the *Win64 - Generic* download.

Download the zip file and extract the single file it contains (curl.exe) to somewhere convenient, such as the desktop. Now start a command prompt by clicking the Start menu followed by the *Run...* option and enter *cmd*.

This will open a command window like the one shown in Figure 5-1. Change directory to wherever you saved curl.exe (in this case Desktop) and then run the command curl just to check that it's ready to use.



```
C:\Documents and Settings\Simon>cd Desktop

C:\Documents and Settings\Simon\Desktop>curl
curl: try 'curl --help' or 'curl --manual' for more information

C:\Documents and Settings\Simon\Desktop>_
```

Figure 5-1. *Figure 5-1. Running curl on Windows*

The command curl is used to send HTTP requests without having to use a browser. You will use this tool to sent commands over the Internet to your Photon/Core in the next project.

Project 6. Control an LED Over the Internet

This project just turns the LED attached to pin D7 of your Photon/Core on or off in response to commands over the Internet. Although in this case, its just an LED, the basic principal is pretty powerful, with the right hardware, you could be turning anything on or off.

The program for this app is called p_06_LED_function. The easiest way to get this onto your Photon/Core is to click on the Libraries button in the Web IDE and then find the library called PHOTON_BOOK. You will find all the project programs used in this book as examples.

Select the file p_06_LED_action and then click the button USE THIS EXAMPLE. A copy of the program will open up in the editor and you can then flash it to your Photon/Core. Note that this will be your own copy of the original program, so you can modify it if you wish.

Software

The code for p_06_LED_Function is as follows.

```
int ledPin = D7;

void setup() {
    pinMode(ledPin, OUTPUT);
    Spark.function("led", ledSwitcher);
}

void loop() {
}

int ledSwitcher(String command) {
    if (command.equalsIgnoreCase("on")) {
        digitalWrite(ledPin, HIGH);
        return 1;
    }
    else if (command.equalsIgnoreCase("off")) {
        digitalWrite(ledPin, LOW);
        return 1;
    }
    return -1;
}
```

The code first defines a variable *ledPin* for the pin D7. This is the pin on the Photon/Core that has a tiny blue LED next to it. The *setup* function then defines this *ledPin* to be an OUTPUT.

The second line of setup is interesting:

```
Spark.function("led", ledSwitcher);
```

This is the line that defines the Function, giving it the name *led*. The second parameter *ledSwitcher* is the name of the C function in the program to be run when the *led* Function is invoked over the internet.

The *loop* function is completely empty. You could do something in here if you wanted, for example make a different LED blink, but it is perfectly ok to leave it empty. This program will only do something when the Function command is received.

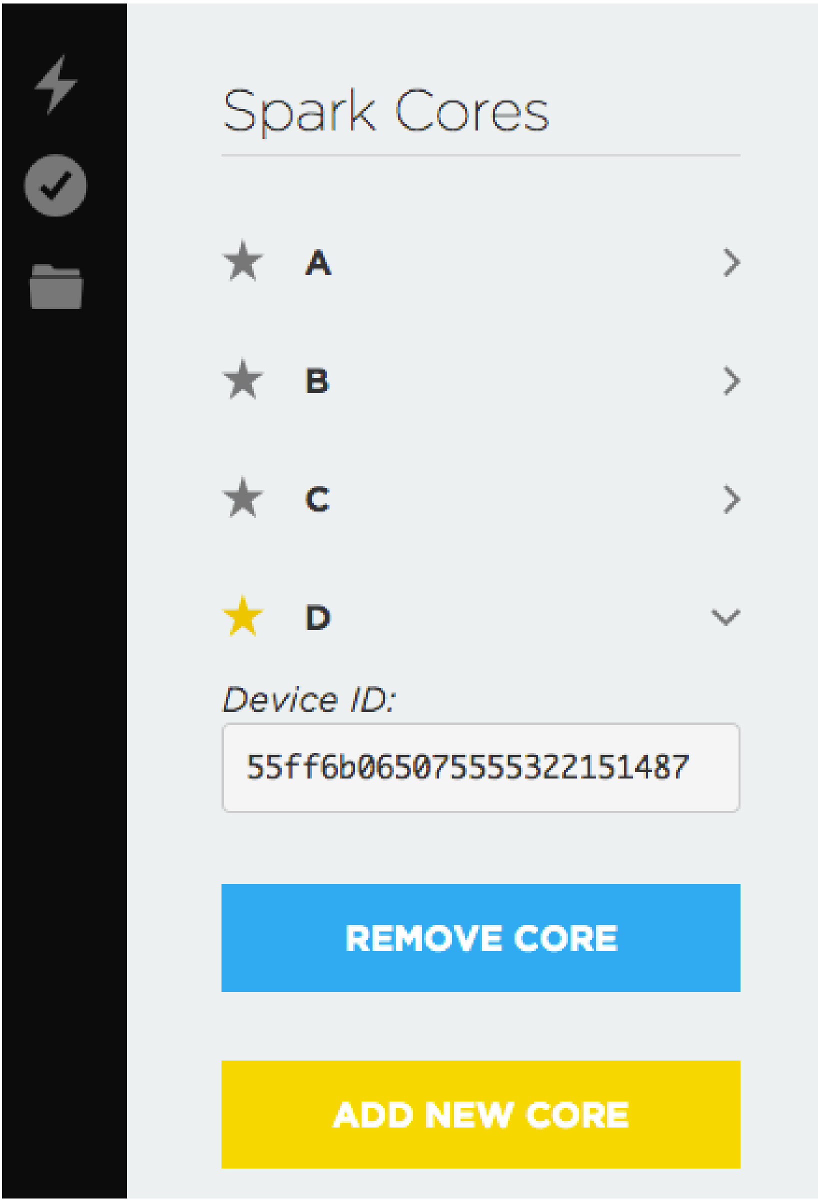
When the *ledSwitcher* function is called as a result of a message arriving, it receives a String as its parameter. In the HTTP request that we are going to send, we will make sure that we either send “on” to turn the LED on or “off” to turn it off. Therefore the ledSwitcher function needs to check the value of the command parameter and perform the appropriate digital write to the pin connected tot eh LED. If the command is either “on” or “off” or for that matter versions of those words in any letter case, the function will then return 1 indicating success. Any other value of command will result in the value -1 being returned by the function.

The return value will actually be returned as part of the response to the HTTP request and this provides valuable information about whether the Function worked or not.

Security

Obviously, it would be potentially dangerous to allow every Spark user to have access to everyone’s Photons. For this reason, there are two long tokens that help secure your Photon.

The first token is a device ID for the Photon or Core. Each of your devices will have a different device ID. To find the device ID for your device, click on the Core’s button on the IDE and select the device whose ID you want. The ID of that Photon/Core will then be displayed. Note that my devices are called A, B, C and D.



Spark Cores

A

B

C

D

Device ID:

55ff6b065075555322151487

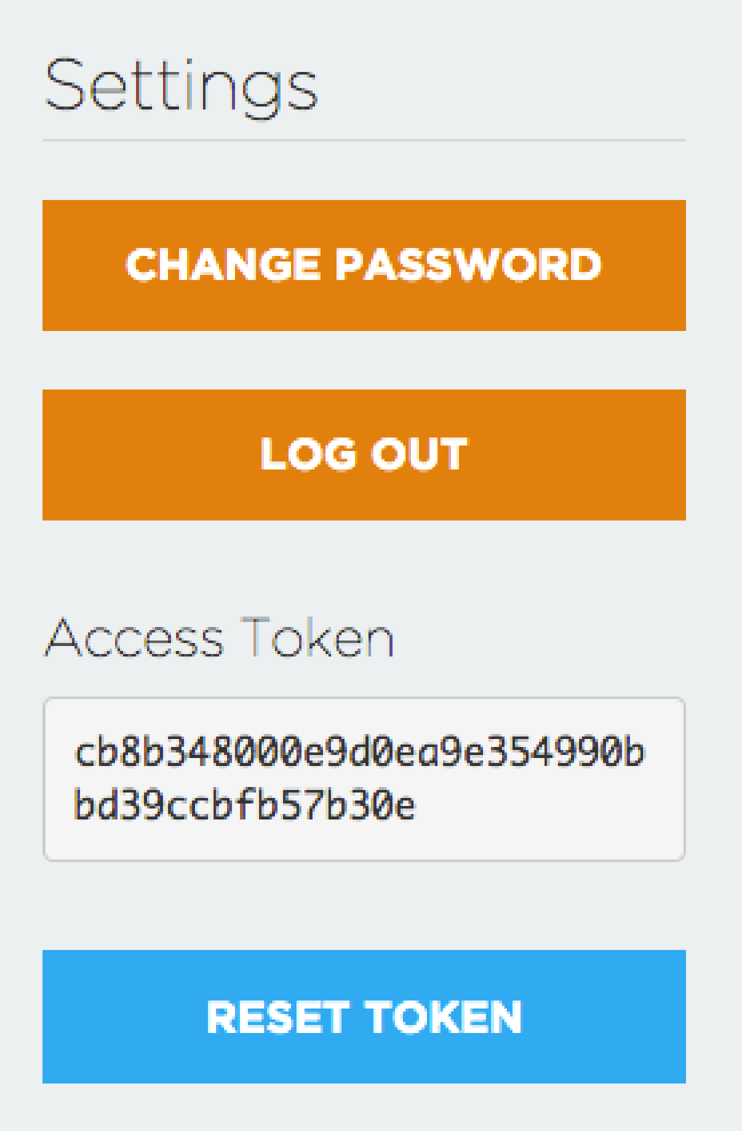
REMOVE CORE
ADD NEW CORE

Figure 5-2. *Figure 5-2. Finding Your Device ID*

To invoke a Function on that Photon/Core, you are going to need to know this ID. A good place to keep this temporarily is as a comment after the last line of your program. For example:

```
// Device ID=55ff74062678501139071667
```

The second token is not specific to a Photon/Core, but rather to your whole Spark account. You will find this if you click on the Settings button on the Web IDE, as shown in Figure 5-3.


Settings
CHANGE PASSWORD
LOG OUT
RESET TOKEN

Access Token

`cb8b348000e9d0ea9e354990bbd39ccbfb57b30e`

Figure 5-3. *Figure 5-3. Finding Your Access Token*

You can at any time generate a new Access Token by clicking on the Reset Token button. In any case, the Access Token will expire after 90 days. (This may change.)

You will also need this token when sending web requests, so you can paste another comment line with it into your program.

```
// Device ID=55ff740626785011390716 67
// Access Token=cb8b348000e9d0ea9e354990bbd39ccbfb57b30e
```

Trying it Out

Now that you have the program running on the Photon/Core, it will just be sitting there waiting for a *led* Function to arrive. Let's give it what it wants by sending an HTTP request to the Spark web service. To do this, we will need to use the two tokens that we just discovered.

If you fetched p_06_LED_Function from the PHOTON_BOOK examples library, then you will see these lines at the end of the program:

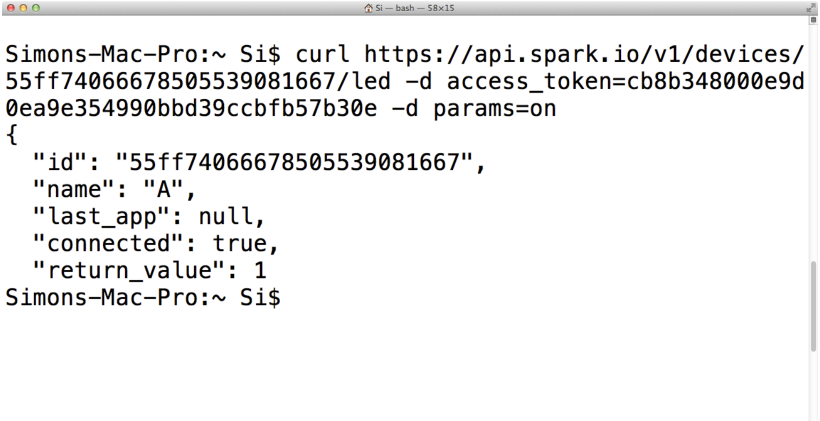
```
// To test with curl
// curl https://api.spark.io/v1/devices/<deviceid>/led
-d access_token=<accesstoken> -d params=on
```

These lines tell us how to use curl to send an HTTP request to control the LED. But first you need to substitute your device ID and access token into the curl command. Once you have changed the comment command to include your tokens, it will look something like this:

```
// curl https://api.spark.io/v1/devices/
55ff74062678501139071667/led
  -d access_token=cb8b348000e9d0ea9e354990bbd39ccbfb57b30e
  -d params=on
```

Note that I have split this over three lines for readability, but the command that you paste into your command line must all be on one line.

Paste the curl command (don't include the *//* comment characters at the start) onto your command line as shown in Figure 5-4.



```
Simons-Mac-Pro:~ Si$ curl https://api.spark.io/v1/devices/
55ff740666785055390816 67/led -d access_token=cb8b348000e9d
0ea9e354990bbd39ccbfb57b30e -d params=on
{
  "id": "55ff740666785055390816 67",
  "name": "A",
  "last_app": null,
  "connected": true,
  "return_value": 1
Simons-Mac-Pro:~ Si$
```

Figure 5-4. *Figure 5-4. Controlling an LED with curl*

If all is well the tiny blue LED on your Photon/Core should light up almost immediately and you will see a response from your device containing the information that the *return_value* is 1.

To turn the LED off again, issue the same command (use up arrow on your keyboard to recall the last command) and change *on* to *off* then press *ENTER* to run the modified command.

You'll probably want to repeat this a few times, because its pretty cool.

Interacting with loop

Now that Project 6 is working in its most basic form, lets make a small change to the program, so that instead of just turning the LED on and off, the *led* Function will set the LED blinking or stop it blinking. You can find the app code for this in the book examples with the name P_06_LED_FUNCTION_BLINK.

```
int ledPin = D7;
boolean blinking = false;
void setup() {
    pinMode(ledPin, OUTPUT);
    Spark.function("led", ledSwitcher);
}

void loop() {
        if (blinking) {
            digitalWrite(ledPin, HIGH);
            delay(200);
            digitalWrite(ledPin, LOW);
            delay(200);
        }
    }

    int ledSwitcher(String command) {
        if (command.equalsIgnoreCase("on")) {
            blinking = true;
            return 1;
        }
        else if (command.equalsIgnoreCase("off")) {
            blinking = false;
            return 1;
        }
        return -1;
    }
```

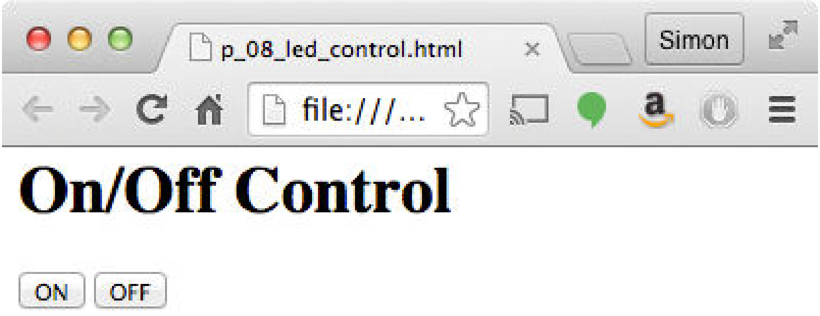
This program has some code in the *loop* function. The code uses the *flag* variable called blinking. If blinking is set to true, the the LED will blink once. This will keep happening as long as blinking is true.

Now, the *ledSwitcher* function does not directly turn the LED on or off, but instead changes the value of the *blinking* flag.

Flash the program onto your Photon/Core and try issuing the same curl commands as you did for the previous program. Now the commands will start and stop the LED blinking rather than simply turn it on or off.

Running Functions from a Web Page

Although curl is good for testing out Functions, its in no way a slick user experience. What would be much nicer would be a web page that has buttons on it, so that when we click a button it controls the LED (see Figure 5-5).



On/Off Control

Figure 5-5. *Figure 5-5. Controlling an LED from a Webpage*

There is a large Javascript library from Spark that encompasses everything from logging in and claiming devices through to calling Functions. You can learn more about this from the Spark documentation here: http://docs.spark.io/javascript/, but for now, to keep things simple, let's use the more familiar jQuery Javascript library on a web page to send HTTP posts in much the same way as when we were using curl, but with a more pleasant user interface.

Start by re-flashing p_06_LED_function onto your Photon/Core. This is the program that simply responds to the "led" Function by either turning the tiny blue LED on D7 on or off, depending on the value of its parameter.

Although eventually, you might want to host the controlling page on a web server somewhere, you can test out the program just with an HTML file on your computer and your browser. The HTML page for this can be found in the "html" section of the downloads for this book available from the book's Github repository at https://github.com/simonmonk/photon_book

Download the file from Github onto your own computer. If you are familiar with GitHub, then you may just want to clone the repository onto your computer to access all the example code in one go. If you do not have git installed on your computer and don't want it, then you can just download a zip archive of all the example code by going to the URL above and then clicking on the Download ZIP button at the bottom right of the webpage.

However you did it, find the ch_05_led_control.html file and open it in a text editor. You also need to replace the values near the top for *accessToken* and *deviceID* with your own access token and device id.

Save the changes to the file and then open it in your browser by double-clicking on it. It should look just like Figure 5-5.

When you click on the ON button it should light the tiny blue LED and when you click on OFF, not surprisingly, it should turn the LED off.

Remember, the code running on the Photon/Core has not changed. All we have done here is use an HTML page to send the HTPP post when we press a button rather than use curl.

The HTML for the page is listed below:

```
<html><head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/
jquery.min.js" type="text/javascript" charset="utf-8"></script>
<script>
var accessToken = "cb8b348000e9d0ea9e354990bbd39ccbfb57b30e";
var deviceID = "55ff74066678505539081667";
var url = "https://api.spark.io/v1/devices/" + deviceID + "/
led";

function switchOn(){
    $.post(url, {params: "on", access_token: accessToken });
}
function switchOff(){
    $.post(url, {params: "off", access_token: accessToken });
}
</script></head>

<body>
<h1>On/Off Control</h1>
<input type="button" onClick="switchOn()" value="ON"/>
<input type="button" onClick="switchOff()" value="OFF"/>
</body></html>
```

If you have never seen HTML before then take a look at an HTML primer such as http://www.w3schools.com/html/html_intro.asp

The HTML file listed above contains a mixture of HTML itself, including text that will appear on the page such as the heading and button labels. It also contains some code written in the JavaScript programming language.

Javascript programs are often embedded in a web page to give it some intelligence, or fancy user interface effects. In this case, the Javascript is going to send the web requests to the Spark web service and hence to your Photon/Core.

If you have done any JavaScript programming in a web setting, you are likely to have come across the jQuery JavaScript library. This library includes loads of useful features for manipulating web pages. It also has a feature that allows it to make HTTP posts, which is just what you need to talk to your Photon/Core.

The jQuery library is imported from one of its homes on the Internet, and then we come to a second section of JavaScript, where some variables are defined for the access token and device ID. You should have already changed these values. The third variable *url* is constructed by splicing the value of *deviceID* into the middle of the url.

Next there are two nearly identical JavaScript functions. As you can see, JavaScript uses the same curly braces for blocks of code and has functions like C but with a different syntax.

If you look at the switchOn function, you can see that it contains the following.

```
$.post(url, {params: "on", access_token: accessToken });
```

The $ symbol is how you access the jQuery library functions, one of which is called *post*. It takes two parameters: the url and what looks like another block of code. This is a Javascript object and is a way of grouping together two values associated with names. The first value of “on” has the name *params* and the second links the value of the *accessToken* variable with the name *access_token*.

The names *params* and *access_token* will become post data to the HTTP post command. These are equivalent to the -d options used in curl.

The body section of the HTML file is responsible for actually displaying the heading and buttons on the web page. Each of the input buttons has an *onClick* event. When you click on one of the buttons it runs the JavaScript function *switchOn* or *switchOff* depending on which button you click.

Security

Now that you can create a webpage on your local hard drive to control your device, the temptation is to place the HTML file onto a web server, so that it can be accessed from anywhere.

You can do this, but not without compromising your security. Anyone doing “View Source” would immediately get access to your Device ID and Access Token. Its just there in the web page for all to see.

If this matters to you, then you should host the page behind a login screen, or on a web server that just runs within your network.

Project 7. Control Relays from a Webpage

In this project, you will build on what you have learned about using Functions, using web pages to control a relay. The software for the project is very similar the the LED example that we have just been through. There is just a bit of renaming of code, to make it more specific to relays rather than LEDs, and the code is extended so that all four relays on the relay shield can be controlled.

The real difference to the project is that it uses a Photon/Core attached to a relay shield. This will allow us to switch all sorts of things on and off.

AC Can Kill You

Although the relays on the relay board are technically capable of switching 120V AC at 10A, you should not use the board to control AC devices unless you are qualified to do so.

Some of the screw terminals on the board will be "hot" (also called "live") if connected to AC, so anything made using this module that does control AC must be properly boxed and earthed so that there is no chance of any exposed metal being touched by unwary fingers.

Be warned every year hundreds of people in the US alone die accidentally from 120V AC electrocution, not to mention the many more people who receive serious burns.

You can still have lots of fun with the relay module, but I strongly suggest that you stick to using it to switch low voltage DC devices.

Figure 5-6 shows a Photon plugged on top of a relay shield that is being used to turn any of the four relays on the module on or off. Note that one of the relays has an old-fashioned 12V electric bell attached to it. Figure 5-7 shows a simple web interface for controlling the relays.

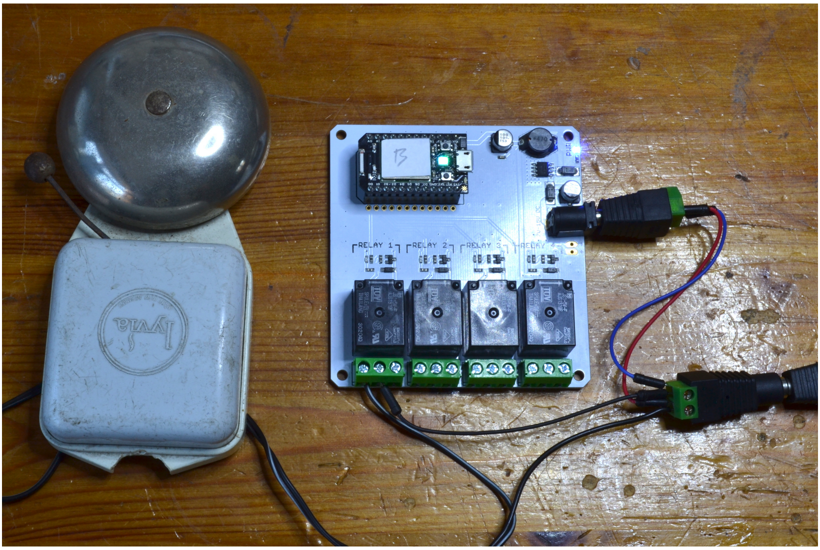


Figure 5-6. *Figure 5-6 Photon Relay and Electric Bell - Good and Noisy!*

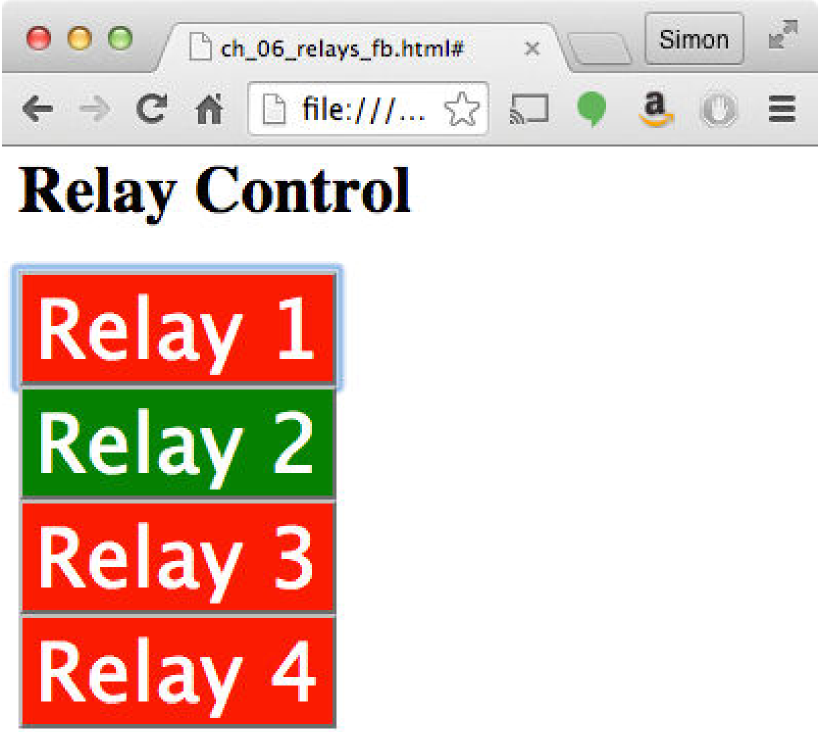

ch_06_relays_fb.html#
Simon
file:///...
Relay Control
Relay 1
Relay 2
Relay 3
Relay 4

Figure 5-7. *Figure 5-7. A Webpage for Controlling Relays*

Parts

To build this project, you are going to need the following parts in addition to your Photon/Core.

Table 5-1. *Parts Bin*

Part	Description	Appendix A code
Relay Shield	Spark shield to control 4 relays	M1
12V Power Supply	12V 1A Power supply	Q1
J1	DC jack to screw adapter (female)	H1

J2	DC jack to screw adapter (male)	H2
Bell	12V DC electric bell	Q2
Wire	Short lengths of multi-core wire.	H3

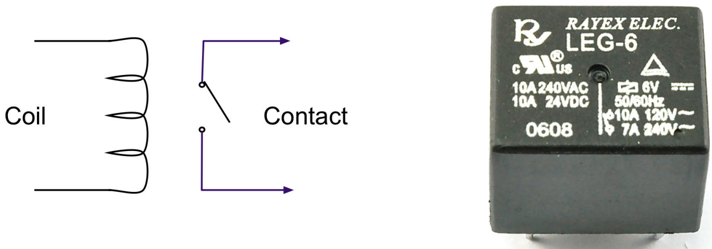
These days, most door bells are of the electronic variety, rather than a good old-fashioned bell complete with a solenoid driven sounder. You should still be able to find such things in your local hardware store, though. Look for something on the packaging that tells you the current that the bell is likely to consume and make sure that the power supply you get will be sufficient to provide this current plus 0.2A for the Photon/Core and relay shield. Most bells should be fine. The extremely noisy bell I used consumed much less than 1A.

Many intruder alarm sirens and strobe lights also operate from 12V, so this may be an interesting alternative to the electric bell.

You can use male-to-male jumper wires for some of the connections to the bell and relay shield, but these tend to have low current-handling capabilities; so some multi-core hookup wire or twin bell wire pulled apart into two strands is better. If you get into making electronics, you will find yourself scavenging odd bits of wire to keep for situations like this. You can also buy hookup wire in various colors.

Relays

A relay (Figure 5-8) is an electromechanical switch. Its very old technology, but reliable and easy to use.



Relay Schemtic

A Relay

Figure 5-8. *Figure 5-8. Relays*

A relay is basically an electromagnet that closes switch contacts. The fact that the coil of the electromagnet and the contacts are electrically isolated from one another makes relays great for things like switching AC-powered devices on and off from digital outputs like those of the Photon/Core. The coil of a relay takes just too much current to be driven directly from the digital output of a Photon, so the relay shield also has a small transistor to switch the relay coil. You will learn more about transistors in Chapter 12.

Whereas the coil of a relay is often energized by between 5V and 12V, the switch contacts can control high power high voltage loads. For example, the relay photographed in Figure 6-7 claims a maximum current of 10A at 120V AC as well as up to 10A at 24V DC.

Relays often have three switch terminals, COM (common), N.O. (normally open), N.C. (Normally Closed). When the relay coil is activated, the COM and N.O. terminals will be connected to each other. When there is no power to the relay coil, the N.C. and COM terminals will be connected. So, generally you will want to use the N.O. and COM connections to switch whatever you are controlling, so that when the power to the Photon/Core is off, the device being switched will also be off.

Design

Figure 5-9 shows how you might wire up a 12V electric bell. You could also substitute 12V DC lighting or any other kind of 12V equipment for the bell. Just make sure that if the thing you are connecting has a positive and negative connection, then the positive connection goes to the power supply and the negative connection to the relay. In this case, the bell does not have a positive and negative.

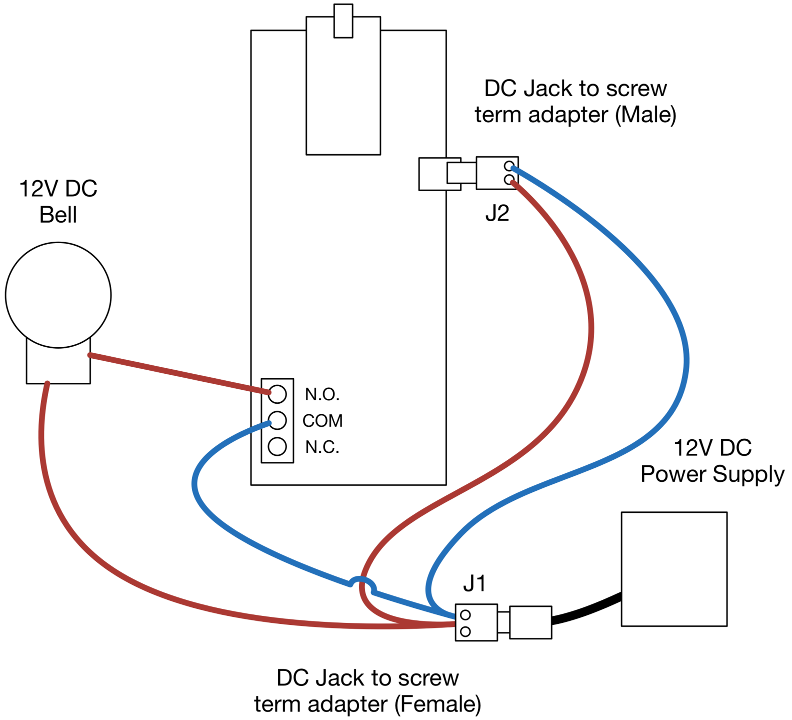

DC Jack to screw
term adapter (Male)
12V DC
Bell
J2
N.O.
COM
N.C.
12V DC
Power Supply
J1
DC Jack to screw
term adapter (Female)

Figure 5-9. *Figure 5-9. Wiring an Electric Bell to a Relay Board*

As you can see, the 12V power supply provides power to both the Relay board and one side of the relay contacts that will switch the electric bell.

When the relay activates, the COM and N.O. (Normally Open) contacts of the relay will be connected together completing the circuit from the bell to the 12V power supply, making the bell sound.

Construction

This is an easy project to make: no soldering is required, and everything is hooked up using screw terminals. Use Figure 5-9 as a reference while you wire things up.

Step 1. Test and Wire the Bell

If the bell does not have flying leads, you will have to attach leads to it. This will depend on your bell, but it may well have screw terminals too. Having attached wires to it, it is not a bad idea to connect the two wires directly to the power supply, just to check that you have the right connections and that the power supply and bell are compatible.

Step 2. Complete the Wiring

The remainder of the wiring centers around the J1 adapter. The tricky bit here is that each of the screw connectors on the adapter must hold two wires. The positive connection on J1 must have one wire going off to the positive connection of J2 and another wire to the bell. The negative terminal of J1 must contain one lead to the negative connection on J1 and another wire to the COM screw terminal of relay 1.

Be very careful to make sure that the positive (+) connections of J1 and J2 are connected together.

Step 3. Plug in the Power Supply

Plug the DC jack from the power supply into J1 and switch it on. If you have a smartphone, you can use the Tinker App to turn pin D0 on and off. This should turn the bell on and off. Pin D0 is connected to Relay 1 on the Relay Shield.

Now you are ready for the software!

Software

The software for this project is in two parts, the app to be run on the Photon/Core and the HTML web page containing Javascript commands that will communicate with the Photon/Core.

Photon Software

The code for the Photon/Core is remarkably compact. The full listing is given below. You can find this code along with all the code used in the book. See the introduction to project 6 for instructions on getting hold of this.

```
int relayPins[] = {D0, D1, D2, D3};
void setup() {
    for (int i = 0; i < 4; i++) {
        pinMode(relayPins[i], OUTPUT);
    }
    Spark.function("relay", relaySwitcher);
}

void loop() {
}

int relaySwitcher(String command) {
    if (command.length() != 2) return -1;
    int relayNumber = command.charAt(0) - '0';
    int state = command.charAt(1) - '0';
    digitalWrite(relayPins[relayNumber-1], state);
    return 1;
}
```

The first interesting thing about this program is that it uses an array to access the four digital pins used to control the relays. The pins that control the relay are D0 to D3, where D0 controls relay 1, D1 relay 2 and so on.

The *setup* function uses a loop to initialize all four pins as outputs and then defines a Spark Function called *relay* that will invoke the C function *relaySwitcher*.

The *relaySwitcher* function is able to switch any of the four relays on and off, not just the one we have connected to a bell. To do this, it uses its *command* parameter. The *relaySwitcher* function expects the *command* parameter to be exactly two characters long, so the first thing the function does is to check that this is the case. If *command* isn't two characters long, then -1 is returned to indicate an error.

Assuming that the *command* parameter is two characters, then the first character will be the number of the relay to control (1 to 4) and the second character will be 1 to turn the relay on or 0 to turn it off.

The code uses a useful trick to convert the character value of the relay number which is of type *char* to an *int*.

```
int relayNumber = command.charAt(0) - '0';
```

The trick is to subtract the character value of "0" from the character value of the first character in the command (command.charAt(0)).

The *digitalWrite* command uses the array and the relay number to decide which pin to set to either 0 or 1 from the value passed in the second character of the command.

Webpage Software

When it comes to the webpage software to control the relays, we will start gently, with a modified version of the HTML and Javascript that we used in the section Running Functions from a Webpage. The problem with that solution is that if something goes wrong, perhaps the Photon/Core isn't connected to the network, then there is no way of knowing from the webpage whether the command was successful and the relay actually turned on.

To get around this problem, there will be a second more complicated version of the webpage that uses the response from the device to provide feedback on the success or failure of the Function command.

Both versions are available from the book's Github page at https://github.com/simonmonk/photon_book in the "html" section of the repository. The first version (listed below) is called ch_06_relays.html and the improved version is called ch_06_relays_fb.html.

```
<html><head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/
jquery.min.js" type="text/javascript" charset="utf-8"></script>
<script>
var accessToken = "cb8b348000e9d0ea9e354990bbd39ccbfb57b30e";
var deviceID = "54ff72066672524860351167";

var url = "https://api.spark.io/v1/devices/" + deviceID + "/
relay";
function setRelay(message){
    $.post(url, {params: message, access_token: accessToken });
}
</script>
</head>
<body>
<h1>Relay Control</h1>
<table>
<tr>
<td><input type="button" onClick="setRelay('11')"
                value="Relay 1 ON"/></td>
<td><input type="button" onClick="setRelay('10')"
                value="Relay 1 OFF"/></td>
</tr>
<tr>
<td><input type="button" onClick="setRelay('21')"
                value="Relay 2 ON"/></td>
<td><input type="button" onClick="setRelay('20')"
                value="Relay 2 OFF"/></td>
</tr>
<tr>
<td><input type="button" onClick="setRelay('31')"
                value="Relay 3 ON"/></td>
<td><input type="button" onClick="setRelay('30')"
                value="Relay 3 OFF"/></td>
</tr>
<tr>
<td><input type="button" onClick="setRelay('41')"
                value="Relay 4 ON"/></td>
<td><input type="button" onClick="setRelay('40')"
                value="Relay 4 OFF"/></td>
</tr></table>
</body></html>
```

The first part of the webpage is exactly the same as the example from Running Functions from a Webpage. Variables are defined for the accessToken and deviceID that you will need to change to match your own access token and device ID.

The separate functions that turned the LED on and off are replaced by a single Javascript function called *setRelay*. The *setRelay* function is passed as a parameter called *message* that will contain the two character code for controlling the relay. The first character being the relay number and the second 1 or 0 to indicate on or off.

The buttons in the user interface call *setRelay* with the right value for the button being pressed.

After editing the deviceID and accessToken variables for your values, open the file in your browser by double clicking on it. The browser window should look like Figure 5-10.

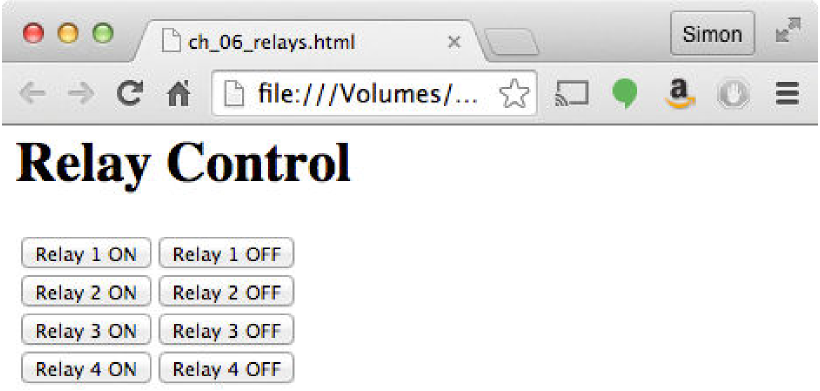


Figure 5-10.
Figure 5-10. A Basic Web Interface for Relays

When you click on the Relay 1 ON button, the relay should click, the little light next to it come on and most importantly the bell will sound. You'll probably want to turn it off again quickly, so click on Relay 1 OFF.

Try the other buttons to make sure that you can control all the relays.

One snag with this interface to the relays is that if you turn off the power to the relay shield and hence the Photon/Core, the webpage will carry on behaving exactly as it currently does, without any indication that nothing is actually happening at the other end of the connection.

This second version of the web page, which is the interface shown in Figure 5-7, has a single button for each relay. Clicking on the button toggles the relay for that button on and off, and changes the color of the button to indicate that the relay is on. However, the button does not change color to indicate that the relay is on until it receives confirmation from the Photon/Core that the Function worked successfully.

The code for this webpage (ch_06_relays_fb.html) is not listed in full below as it's some 60 lines long. You may wish to have the file open in an editor while the code is explained.

```
<style type="text/css">
      .on  { background:green; }
      .off { background:red; }

       input[type=button] {
         color: white;
         font-size: 2.5em;
      }
</style>
```

The *style* block defines the colors to be used with the button for its on and off states and also increases the size of the font on the buttons.

Before looking at the Javascript function *toggleState*, where all the action happens, it is perhaps worth looking at how it is called from one of the buttons:

```
<input type="button" id="btn" value="Relay 1" class="off"
       onclick="toggleState(this, '1')" />
```

When a button is clicked, *toggleState* is called with two parameters. The first parameter is *this*, which refers to the button being pressed. The *toggleState* function will need this to determine the current state of the button. The second parameter is the relay number as a string.

The *toggleState* function is listed below:

```
function toggleState(item, relayNumber){
    function setRelay(message){
        $.post(url, {params: message, access_token: 
accessToken}, callback);
    }
    function callback(data, status){
        if (data.return_value == 1) {
            if (item.className == "on") {
                item.className="off";
            }
            else {
                item.className="on";
            }
        }
        else
        {
            alert("Could not Connect to Photon");
        }
    }
   if(item.className == "on") {
      setRelay(relayNumber + "0");
   } else {
      setRelay(relayNumber + "1");
   }
}
```

In this case, the functions *setRelay* and *callBack* are defined within toggleState, so that they can have access to toggleState's variables, in particular to its *item* parameter. The *setRelay* function doesn't actually need this, but it makes sense to keep both functions together as they are closely related.

The *setRelay* function is very like the earlier version of this function in the simplified version of this page. There is one important difference and that is that the call to *$.post* now includes an additional parameter of *callback*. This is a reference to the other function inside *toggleState*. The *callback* function will automatically be run when the web request to the Photon/Core completes. We will return to the *callback* function shortly, but first let's look at the main body of the *toggleState* function.

```
   if(item.className == "on") {
      setRelay(relayNumber + "0");
   } else {
      setRelay(relayNumber + "1");
   }
```

The *item* variable contains a reference to the button that was clicked and its *className* will either be *on* or *off* depending on the current state of the button. The *setRelay* function is then called, appending 0 or 1 to the relay number as appropriate. Note that at this point the appearance of the button has not changed; this will only happen if the communication with the device is successful.

When the request to the Photon/Core completes, the *callback* function will be invoked. Its first parameter (*data*) will be the response from the Photon/Core via the Spark web service. This is the same response that you are used to seeing as a result of a curl command to invoke a Function. The *callback* function picks out the *return_value* from this response and as long as it is 1, indicating success (-1 indicates failure), the button appearance is changed by setting its *className*.

If *callback* does not get a response of 1, then it alerts an error message saying "Could not connect to Photon."

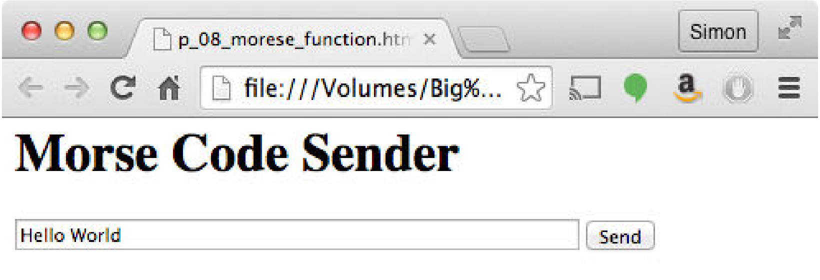
Remembering to change the *accessToken* and *deviceID* variable in your file, open a browser on the page and try turning some relays on and off.

Project 8. Morse Code Text Messages

Projects 5 and 6 both flashed out a Morse Code message, but suffered from the limitation that the message was fixed. The only way to change the message was to alter the value of the message variable in the program and then flash it onto your Photon/Core.

In this project the message to be sent as Morse Code is sent as the parameter to a Spark Function. The maximum length of this parameter is 64 characters, so you will need to keep your message below that number of characters.

The project also has a web page where you can specify the message to send (Figure 5-11).



Morse Code Sender

Hello World

Figure 5-11. *Figure 5-11. Sending Morse from a Webpage*

For added fun the project will also gain a buzzer so that the Morse is both flashed and beeped.

Parts

To build this project, you are going to need the following parts in addition to your Photon/Core.

Table 5-2. *Parts Bin*

Part	Description	Appendix A code
LED1	Red or any other color LED	C2
R1	220Ω resistor	C1
R2	1kΩ resistor	C5
S1	Piezo buzzer	C4
	Male-to-male jumper wires	H4
	Half-sized breadboard	H5

These components are all included in the Maker's Kit and apart from the buzzer are the same as for Project 4.

Software

Most of the code here is our now standard Morse sending code. You can find the full program to flash onto your device under the file name P_08_Morse_Function in the PHOTON_BOOK library in the Web IDE.

The first difference between this and our earlier Morse programs is the addition of a new pin for the buzzer.

```
int buzzerPin = D1;
```

Because sending a Morse message can take a while, it would be better if the handler for the Function to send a Morse message would return immediately with a status number of 1 or -1 rather than wait for the entire message to be sent. That way, the web page from which the message is sent could provide confirmation that the Morse code was sending on the Photon/Core. To allow this to happen there needs to be a new Boolean value that we can set to *true* to indicate that flashing should start.

```
boolean flashing = false;
```

The *setup* function now needs an extra line to initialize this pin as an output. The *setup* function also contains the line to define the Spark Function that will be invoked whenever the Photon/Core receives a Function command to send a message. This will call the function *startFlashing*.

```
void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(buzzerPin, OUTPUT);
    Spark.function("morse", startFlashing);
}
```

The *loop* function will be responsible for doing the flashing, but only if the *flashing* variable is set to true.

```
void loop() {
    if (flashing) {
        flashMessage(message);
        flashing = false;
    }
}
```

The *startFlashing* function handles the Spark Function called *flash*. The *startFlashing* function does not actually do the flashing, instead it first checks that the length is less than or equal to the maximum allowed parameter size of 64 characters. If its not too long, the flashing variable is set to true, so that the main loop can start flashing (and buzzing) the message. If all is good, 1 is returned, otherwise -1 is returned to indicate an error.

```
int startFlashing(String param) {
    if (message.length() <= 64) {
        message = param;
        flashing = true;
        return 1;
    }
    else {
        return -1; // message too long
    }
}
```

The only other change to the code is in the *flash* function that must now buzz at the same time.

```
void flash(int duration) {
    digitalWrite(ledPin, HIGH);
    tone(buzzerPin, 1000);
    delay(duration);
    noTone(buzzerPin);
    digitalWrite(ledPin, LOW);
}
```

Buzzing is accomplished using the tone and noTone commands. These commands can only be used on the pins D0, D1, A0, A1, A4, A5, A6, A7, RX and TX**.**

The function tone takes two parameters, the pin to play a tone on and the frequency of that tone in Hertz (cycles per second). Once started using *tone*, the pin will continue to oscillate until the *noTone* command, with a parameter of that pin, is called.

Flash the program onto your Photon or Core so that the app is installed before you start attaching hardware to it.

You saw the webpage for controlling the Morse sender in Figure 5-11. The code for this is listed below. This can be found with the book downloads in the “html” section of the repository. The file is called p_08_morse_function.html.

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/
jquery.min.js" type="text/javascript" charset="utf-8"></script>

<script>
```

```
var accessToken = "cb8b348000e9d0ea9e354990bbd39ccbfb57b30e";
var deviceID = "54ff72066672524860351167"
var url = "https://api.spark.io/v1/devices/" + deviceID + "/
morse";

function callback(data, status){
    if (data.return_value == 1) {
        alert("Your Message is being sent");
    }
    else
    {
        alert("Could not Connect to Photon");
    }
}

function sendMorse(){
    message = $("#message").val();
    $.post(url, {params: message, access_token: accessToken},
                callback);
}
</script>
</head>

<body>
<h1>Morse Code Sender</h1>
<input id="message" value="Hello World" size="64"/>
<input type="button" value="Send" onclick="sendMorse()" />
<br/>
</body>
</html>
```

The webpage code follows much the same pattern as the previous examples. The url is constructed using the *accessToken* and *deviceID* variables that must be changed to match those of your account and device.

In this example, the callback function does not need to be nested within the *sendMorse* function as it does not need access to any of its variables or parameters. The *callback* function pops up an alert dialog to indicate success or failure.

The *sendMorse* function first extracts the *message* to be sent from the field on the webpage with the id of message and then supplies this as the parameter to the HTTP request that will trigger the *morse* Function on the Photon/Core.

Hardware

Figure 5-12 shows the breadboard layout for the project.

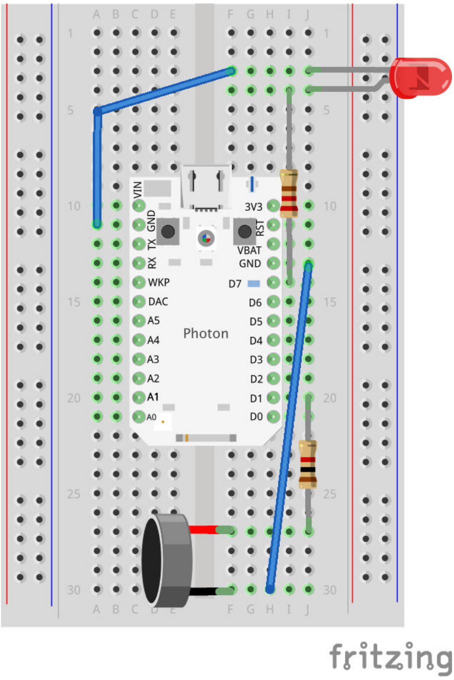

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 5-12. *Figure 5-12. Breadboard Layout for Morse with Buzzer*

You can tell the 220Ω and 1kΩ resistors apart because the 220Ω resistor has stripes that are red, red, brown and the 1kΩ resistor brown, black, red. The buzzer can be placed either way around, but remember that the LED must have the longer positive lead on the same row as the resistor R1.

When the breadboard is fully assembled, power up the Photon/Core, its time to try out the project.

Using the Project

Open the webpage for this project in your browser and enter a message into the text field and press the send button. After a few moments, the Morse code message that you typed will start to be sounded by the buzzer, while at the same time flashed by the LED. Remember that the maximum message length is 64 characters.

Spark Variables

So far, all of our Internet of Things projects have been about telling the Photon/Core to do something using Functions. Being able to read information from the Photon/Core is just as important. Projects 9 and 10 are both concerned with using sensors with a Photon over the Internet.

To fetch information from a Photon/Core you need to use Spark Variables. Defining variables in your app allow you to send requests to your Photon/Core that will bring back a value. Rather like Functions, Spark Variables can easily be confused with regular C variables, so I will use an uppercase **V** when referring to Spark **V**ariables.

Variables have other similarities with Functions. They have an identifier which is a short string used to identify the Variable, so that you know what to ask for in a web request. They are also defined in the *setup* function, but rather than being associated with a C function, they are associated with a C variable, whose value they will report. It is the job of your app to keep updating the values in the C variable, so that when a request comes in for a Variable, the value returned is up to date.

In the next project you will see how this works, in a simple example, but first you need to learn a little about how you can connect sensors to a Photon/Core and take readings from them.

Analog Inputs

You will find six pins down the left hand side of the Photon labelled A0 to A5. The Core actually has two extra analog inputs A6 and A7. Although these pins can be used as digital inputs or outputs just like D0 to D7, the *A* pins are capable of analog input.

Whereas a digital input can only tell if something is on or off, an analog input can measure the voltage at a pin as a number between 0 and 4095, where a reading of 0 indicates 0V and 4095 means 3.3V.

Maximum Voltage of 3.3V
Do not be tempted to attach an analog input to a voltage higher than 3.3V. This is likely to damage your Photon.

If a sensor whose output voltage varies somewhere in the range 0 to 3.3V, you can connect it to one of the analog pins and read the voltage and therefore work out a value for the property that the sensor is measuring.

As an example of this, load up the file “ch_05_analog_ins” from the PHOTON_BOOK example library. This is listed below:

```
int reading = 0;
int analogPin = A0;

void setup() {
    Spark.variable("analog", &reading, INT);
}

void loop() {
  reading = analogRead(analogPin);
}
```

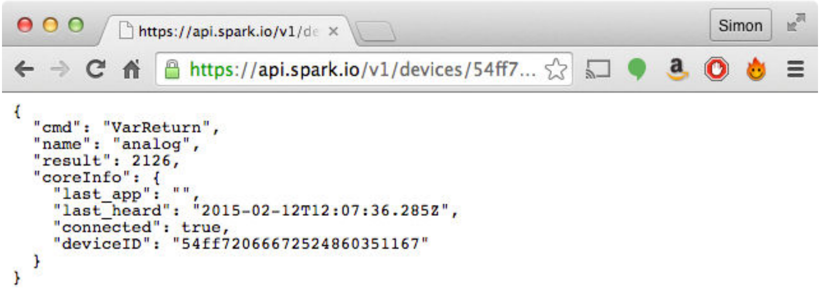
Here the program starts with a C variable (*reading*). This will be the C variable whose value will be returned when a web request Function is sent to the Photon/Core. The other variable (*analogPin*) just identifies the analog pin to use, in this case, A0.

The link between the C variable and the Spark Variable takes place in the *setup* function. The first parameter (“analog”) is the name given to the Variable and the second parameter (&reading) identifies the C variable that the analog Variable is to use. The & symbol is a C language feature that indicates the memory address of a variable. If the second parameter were just *reading* rather than *&reading*, then the value of *reading* would just be substituted into that parameter and since reading is 0 at the time, the Variable values would never change. Using & allows the Photon/Core's firmware to defer fetching the value of the variable until it needs it for the Variable.

Flash the program onto your Photon/Core. To test it, you can just use a web browser, because unlike Functions, Variables use HTTP "get" requests, which you can issue simply by entering a URL into the address bar of your browser. So, open a browser window and enter the URL below into it.

```
https://api.spark.io/v1/devices/<DEVICE ID>/analog?
access_token=<ACCESS TOKEN>
```

Remember to substitute your device ID and access token into the URL. The result from the web page will look like Figure 5-13.


https://api.spark.io/v1/de
Simon
https://api.spark.io/v1/devices/54ff7...

```
{
  "cmd": "VarReturn",
  "name": "analog",
  "result": 2126,
  "coreInfo": {
    "last_app": "",
    "last_heard": "2015-02-12T12:07:36.285Z",
    "connected": true,
    "deviceID": "54ff72066672524860351167"
  }
}
```

Figure 5-13. *Figure 5-13. Requesting a Variable using the Browser*

Refresh the page a few times and you should see the value of "result" change a little each time. This is not really surprising as the analog does not have anything connected to it, it is said to be "floating."

Using a male to male jumper wire, connect A0 first to GND (Figure 5-14a) and then to 3.3V (Figure 5-14b), and each time refresh the web page. You should find that with A0 connected to GND you will get a result of 0 and with it connected to 3.3V something close to 4095.

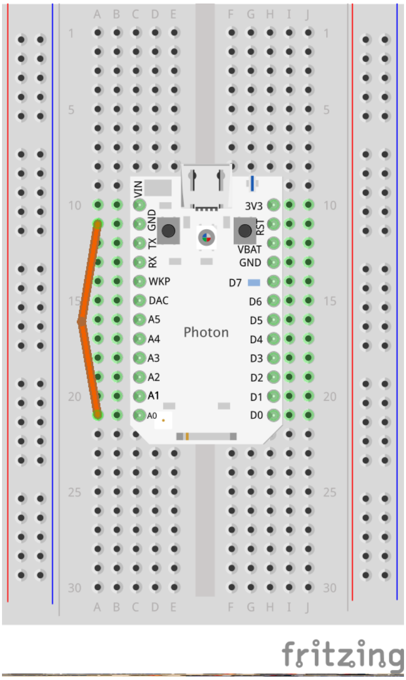

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 5-14. *Figure 5-14a. Connecting A0 to GND*

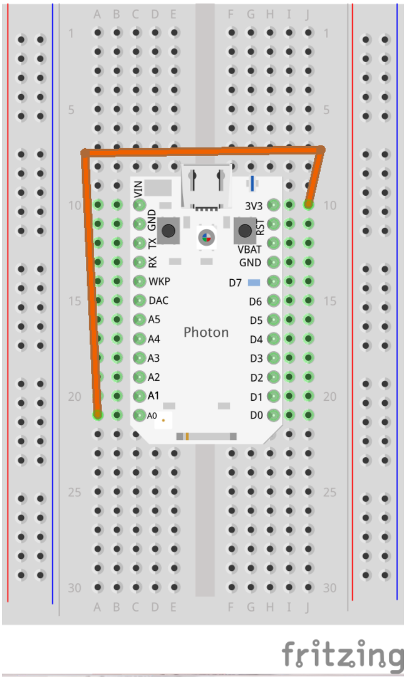

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 5-15. *Figure 5-14b. Connecting A0 to 3.3V*

In project 9, we will use a photoresistor as a light sensor, whose value can be retrieved using a Variable web request.

Project 9. Measuring Light over the Internet

In this project, you will use a photoresistor to measure light intensity and display a graphical representation of the light level on a web page, as the voltage at pin A0 (Figure 5-15).

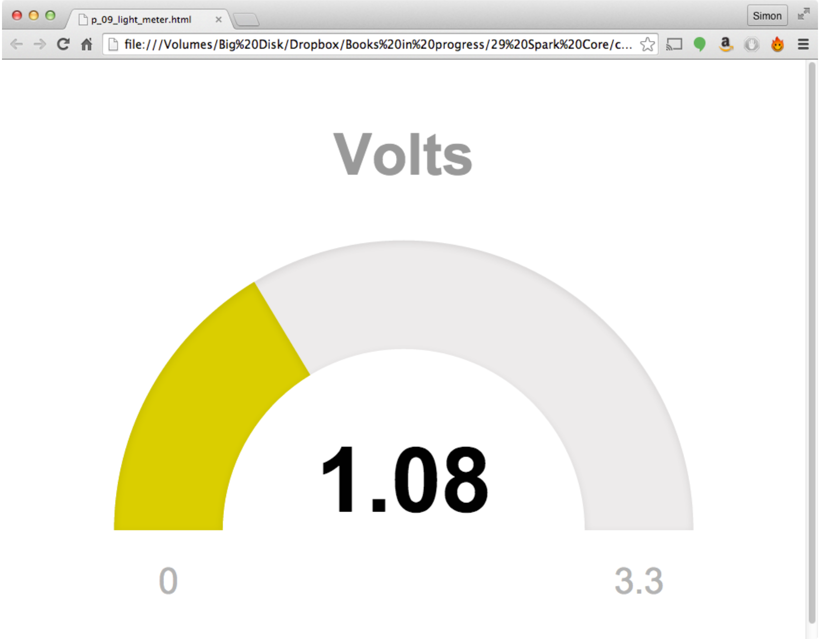

p_09_light_meter.html
file:///Volumes/Big%20Disk/Dropbox/Books%20in%20progress/29%20Spark%20Core/c...
Simon
Volts
1.08
0
3.3

Figure 5-16. *Figure 5-15. Displaying the Voltage at A0*

Parts

To build this project, you are going to need the following parts in addition to your Photon or Core.

Table 5-3. *Parts Bin*

Part	Description	Appendix A code
R1	Photo resistor (1kΩ)	C6
R2	1kΩ resistor	C5
	Male-to-male jumper wires	H4
	Half-sized breadboard	H5

These components may be included in the Maker's Kit.

Software

The app for this project can be found in the file P_09_Lightmeter within the PHOTON_BOOK library in the Web IDE.

```
int reading = 0;
double volts = 0.0;
int analogPin = A0;

void setup() {
    Spark.variable("analog", &reading, INT);
    Spark.variable("volts", &volts, DOUBLE);
}

void loop() {
  reading = analogRead(analogPin);
  volts = reading * 3.3 / 4096.0;
}
```

This code is very similar to that of the example in the previous section. However, this version has two Variables, *reading* and *volts*. In fact, you can have up to 10 Variables in your app.

The *reading* Variable is associated with the *reading* C variable and returns just the same result as the previous example, however, the second Variable (*volts*) will return the actual voltage at the analog pin. To calculate this voltage, the raw reading is first multiplied by 3.3 and then divided by 4096 (the maximum range of reading).

The fancy graphical display is courtesy of a really handy Javascript plugin called JustGage (http://justgage.com/) The webpage for this can be found in the file p_09_light_meter.html along with the JustGage javascript libraries.

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/
jquery.min.js" type="text/javascript" charset="utf-8"></script>
<script src="raphael.2.1.0.min.js"></script>
<script src="justgage.1.0.1.min.js"></script>

<script>
var accessToken = "cb8b348000e9d0ea9e354990bbd39ccbfb57b30e";
var deviceID = "54ff72066672524860351167"
var url = "https://api.spark.io/v1/devices/" + deviceID + "/
```

```
volts";

function callback(data, status){
    if (status == "success") {
        volts = parseFloat(data.result);
        volts = volts.toFixed(2);
        g.refresh(volts);
        setTimeout(getReading, 1000);
    }
    else {
        alert("There was a problem");
    }
}

function getReading(){
     $.get(url, {access_token: accessToken}, callback);
}
</script>
</head>

<body>
<div id="gauge" class="200x160px"></div>

<script>
var g = new JustGage({
    id: "gauge",
    value: 0,
    min: 0,
    max: 3.3,
    title: "Volts"
});
getReading();
</script>

</body>
</html>
```

The JustGage library and the Raphael library that it uses are both imported and variable-defined for the *accessToken* and *deviceID*. Don't forget to change these values for your account and device.

Jumping now to the end of the file, the Javascript function *getReading* is called. This constructs the HTTP request to be sent and attached the *callback* function to it.

When the request responds, the *callback* function checks to see if the request was successful and if it was, it retrieves the voltage reading from the result, converting it from a string to a float using the command *parseFloat*. Because the number may have a lot of digits after the decimal point, the number is truncated to two decimal places using the *toFixed* function.

The gauge display is updated with the new voltage reading by calling g.refresh, after which another call to *getReading* is scheduled for one second later using *setTimeout*.

The gauge control itself requires a *div* tag with an id of gauge where it should draw itself and a short section of script that initializes it. Here you can change the minimum and maximum values for the gauge as well as the title.

Hardware

Figure 5-16 shows the breadboard layout for project 9. Both the resistor and photoresistor can be connected either way around. They form a voltage divider (see side-bar) that changes the voltage at A0 depending on the quantity of light falling on the photoresistor.

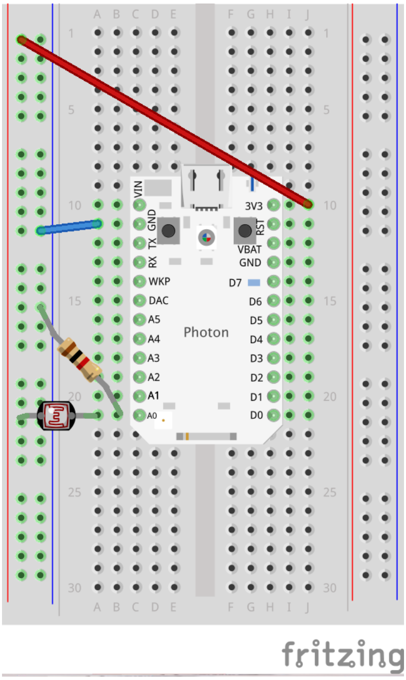

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 5-17. *Figure 5-16. Breadboard Layout for Project 9*

Once you have connected up the breadboard, power up your Photon/Core and the project is ready to use.

Photo Resistors and Voltage Dividers

In this project, a fixed value resistor is used with a photoresistor whose resistance varies according to how much light falls on it. They are in an arrangement called a voltage divider (see Figure 5-17).

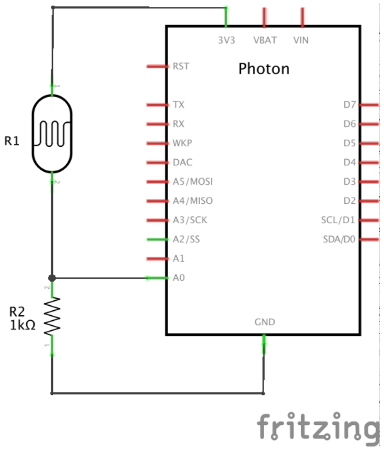


Figure 5-18. *Figure 5-17. Voltage Dividers*

If the photoresistor had just the right amount of light falling on it to make its resistance the same as R2 (1kΩ) then the 3.3V would be split equally across the two resistors and the voltage at A0 would be half of 3.3V (1.66V). The lower the resistance of the photoresistor (more light) the less voltage it will grab and therefore the more voltage there will be across R2 and therefore the higher the voltage at A0.

Excessive Data Polling

Repeatedly reading data in this way is called polling. In this case, the web page is requesting a Variable value from a Photon/Core every second. If you imaging tens or hundreds of thousands of Photon users all doing this kind of thing at the same time, the Spark service is going to be hammered pretty hard.

So bursts of polling once a second while you are using it are fine, but the socially responsible thing to do would be to make sure that you close the browser window when you are not using it, or set the polling rate to say once every 10 seconds if it's going to be running all the time.

Using the Project

Open the webpage p_09_light_meter.html and the window should look like Figure 5-15.

Cover the photoresistor with your hand and you should see the reading fall, hold it up to the light and the meter should swing over to the right.

There are other types of resistive sensor that could be substituted for the photoresistor, including strain gauges, variable resistors (like volume controls) and even some gas detectors.

In the next project, you will use a different type of sensor to provide remote temperature monitoring.

Project 10. Measuring Temperature over the Internet

This project is very similar to the last, but rather than measure light intensity with a photoresistor, it uses a digital temperature sensor chip. The gauge display is perfect for this. Figure 5-18 shows the display in degrees F, but this can easily be changed to degrees C.

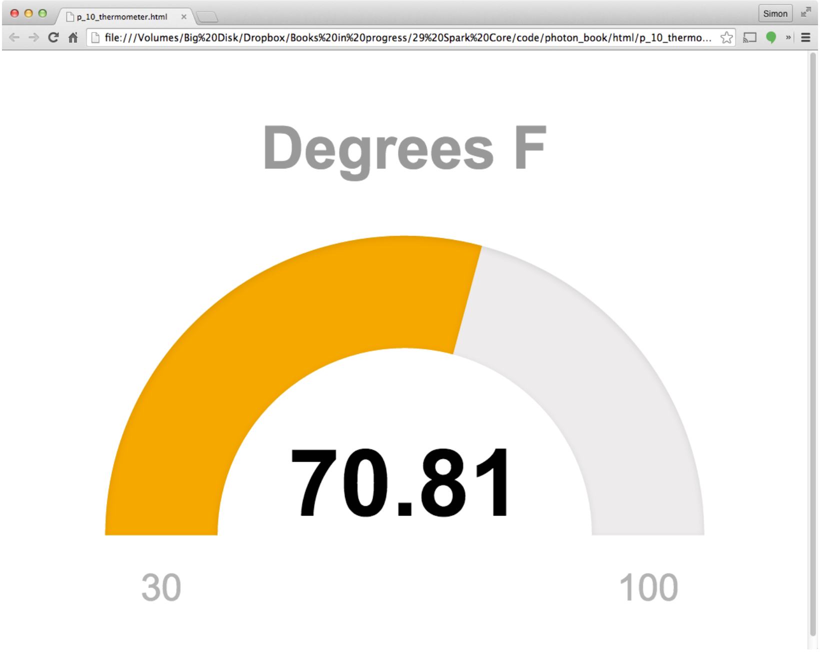

p_10_thermometer.html
Simon
file:///Volumes/Big%20Disk/Dropbox/Books%20in%20progress/29%20Spark%20Core/code/photon_book/html/p_10_thermo...
Degrees F
70.81
30
100

Figure 5-19. *Figure 5-18. An Internet Thermometer*

Parts

To build this project, you are going to need the following parts in addition to your Photon/Core.

Table 5-4. *Parts Bin*

Part	Description	Appendix A code
IC1	DS18B20 temperature sensor	C7
R1	4.7kΩ resistor	C9
C1	100nF (0.1uF) capacitor	C8
	Male-to-male jumper wires	H4
	Half-sized breadboard	H5

These components may be included in the Maker's Kit.

The temperature sensor used is actually a chip enclosed in a water-tight container with a lead attached to it, so you can place the sensor some distance away from the Photon/Core if you want.

Software

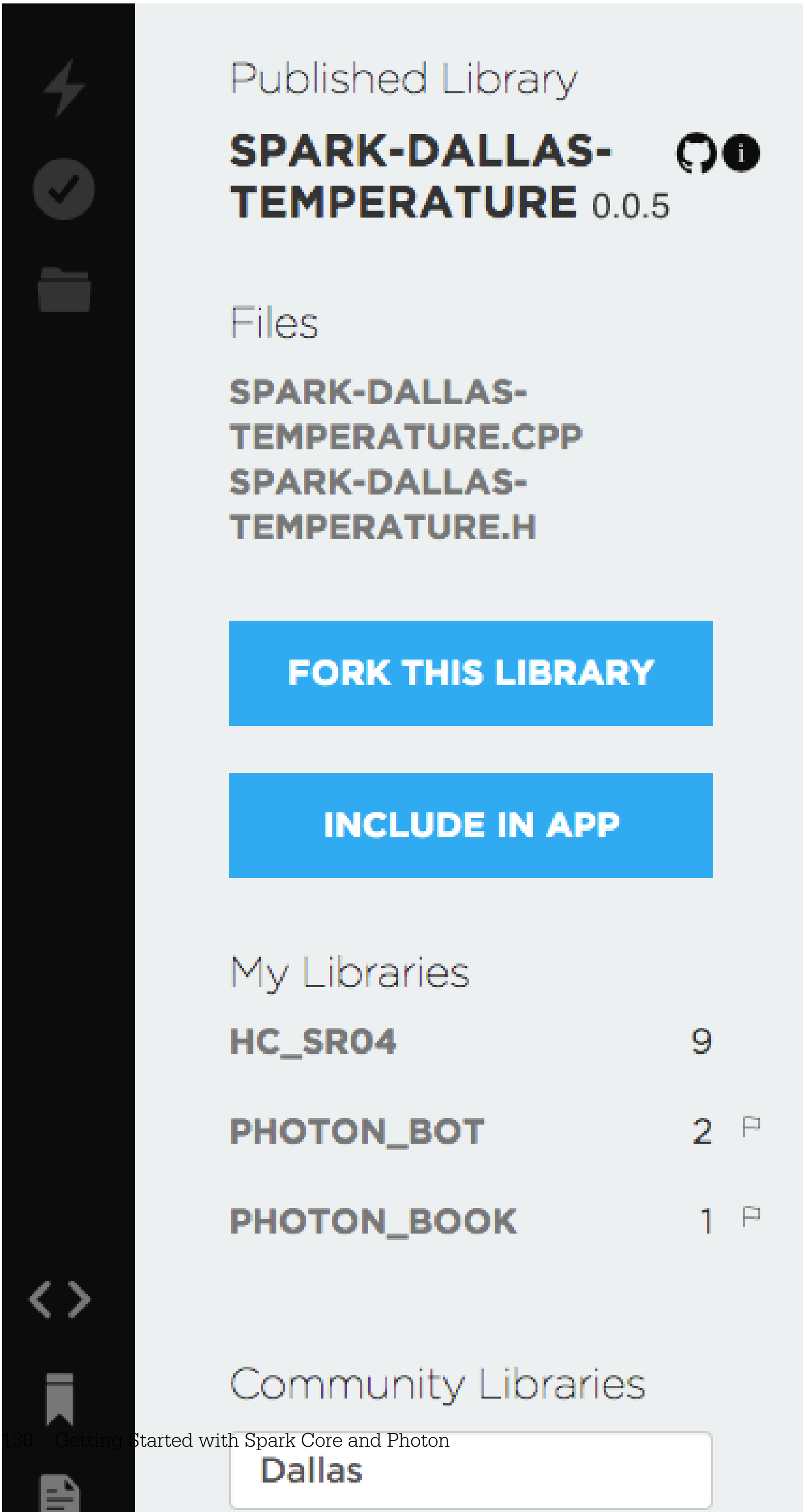
The app for this project can be found in the file P_10_Thermometer_Dallas within the PHOTON_BOOK library in the Web IDE. The name Dallas is used, because this common type is made by Dallas Semiconductors.

The program includes two Variables, one that reports the temperature in degrees F and one in degrees C.

```
#include "OneWire/OneWire.h"
#include "spark-dallas-temperature/spark-dallas-temperature.h"

double tempC = 0.0;
double tempF = 0.0;

int tempSensorPin = D2;

OneWire oneWire(tempSensorPin);
DallasTemperature sensors(&oneWire);

void setup() {
    sensors.begin();
    Spark.variable("tempc", &tempC, DOUBLE);
    Spark.variable("tempf", &tempF, DOUBLE);
}

void loop() {
  sensors.requestTemperatures();
  tempC = sensors.getTempCByIndex(0);
  tempF = tempC * 9.0 / 5.0 + 32.0;
}
```

The first thing to note about this program is that it uses two libraries, indicated by the #include statements at the top. Libraries are a way of sharing code for anyone to use, and, because interfacing to DS18B20 chip is a little tricky, Tom de Boer has converted an original Arduino library to work with the Photon and Core. This library (spark-dallas-temperature) relies on another library called OneWire to handle the serial communication with the temperature sensor chip.

For your app to work, as well as having the #include commands at the top, you also need to add the libraries to the app. To do this, first click on the Library button in the Web IDE and then type "Dallas" into the search box beneath "Community Libraries" (Figure 5-19).



Published Library

SPARK-DALLAS-TEMPERATURE 0.0.5

Files

SPARK-DALLAS-TEMPERATURE.CPP

SPARK-DALLAS-TEMPERATURE.H

FORK THIS LIBRARY
INCLUDE IN APP

My Libraries

HC_SR04	9
PHOTON_BOT	2
PHOTON_BOOK	1

Community Libraries

Dallas

an App

Click on the INCLUDE IN APP button and then scroll down to P_10_THERMOMETER_DALLAS, select it and then click ADD TO THIS APP. Repeat the same process, searching for the library OneWire.

Turning to the code itself, after the variable tempSensorPin has been defined as D2, the definition below is called to start the OneWire serial bus running on that pin.

```
OneWire oneWire(tempSensorPin);
```

The next line instructs the DallasTemperature library to use that interface when communicating to temperature sensors. You can actually connect a whole load of temperature sensors to the same pin if you want.

The *setup* function contains a call to *sensors.begin()* to start the sensors monitoring the temperature.

The *loop* function contains the code to actually get hold of a reading. This is a two-stage process. First *sensors.requestTemperatures* is called that gets readings from all the sensors (even though in this case there is only one). The second step is to access the reading itself using *getTempCByIndex*. The index supplied as a parameter is 0 for the first sensor attached to the Photon/Core. If there was a second DS18B20 attached, it would be index 1 and so on.

The temperature in degrees F is then calculated using a standard formula.

The webpage is almost identical to the light sensor page, but with a few changes to reflect the fact that it is the temperature being displayed. The webpage can be found in the file p_10_thermometer.html.

Hardware

The breadboard layout for the project is shown in Figure 5-20.

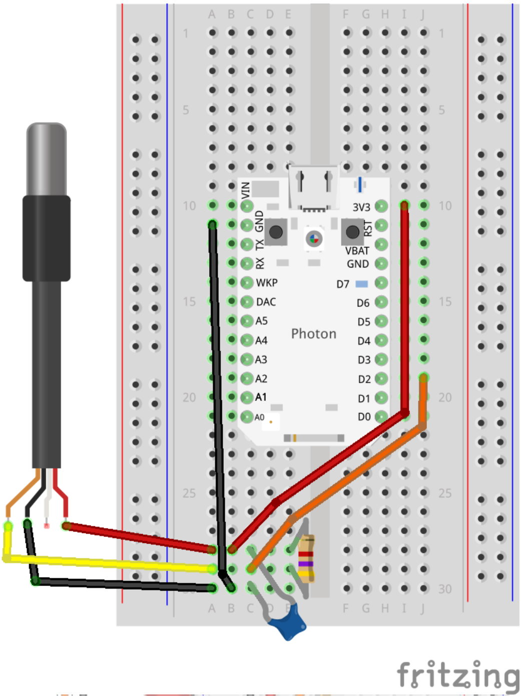

A B C D E
F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
Photon
fritzing

Figure 5-21. *Figure 5-20. Breadboard Layout for Project 9*

The temperature sensor actually has four connectors. One is a connection to the braided screening within the cable. This does not need to be connected. The other wires are red (positive supply), black (GND) and yellow (the data signal). The resistor must be connected between the data signal and positive supply and the capacitor between the positive supply and GND.

Using the Project

To use the project, just upload the app to the Photon/Core and open the webpage p_10_thermometer.html in your browser.

If you want to change the temperature display to be in degrees C rather than F, then open p_10_thermometer.html in an editor and make the following changes:

Change the lines:

```
var url = "https://api.spark.io/v1/devices/" + deviceID + "/
tempf";
```

to read:

```
var url = "https://api.spark.io/v1/devices/" + deviceID + "/
tempc";
```

In other words, use the “tempc” Variable in place of the “tempf” Variable.

You also need to change the title of the gauge and its range, so modify the script block at the end of the file to look like this:

```
<script>
var g = new JustGage({
    id: "gauge",
    value: 0,
    min: 0,
    max: 30,
    title: "Degrees C"
});
getReading();
</script>
```

Reload the page in your browser and it should now look like Figure 5-21.

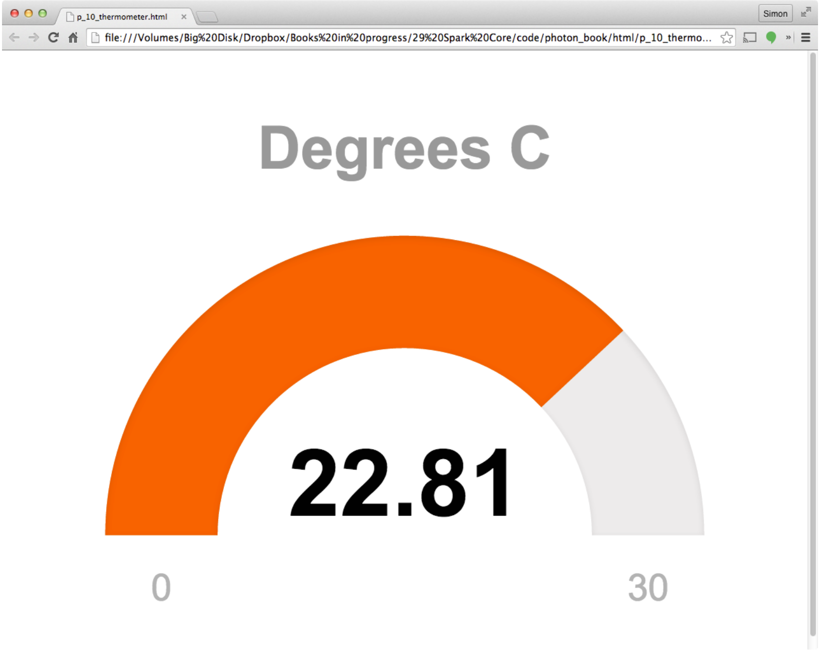

p_10_thermometer.html
file:///Volumes/Big%20Disk/Dropbox/Books%20in%20progress/29%20Spark%20Core/code/photon_book/html/p_10_thermo...
Simon
Degrees C
22.81
0
30

Figure 5-22. *Figure 5-21. The Temperature in Centigrade*

Summary

In this chapter, you have used analog inputs attached to sensors and digital outputs controlling LEDs and buzzers to make some simple IoT projects. In the next chapter you will learn more about the Internet of Things and how to use the Photon and Core with the popular If This Then That (IFTTT web service).

6/If This Then That

The If This Then That (IFTTT) web service allows you link together various internet services in a simple way. IFTTT (http://ifttt.com) allows you to define “recipes" such as “if someone forks one of my Github projects, send me an email, telling me who." IFTTT also allows linking of more conventional web services like Gmail, Twitter, Facebook etc. IFTTT can also integrate with IoT services, including Spark. This means that you can make IFTTT recipes that use a Photon or Core. For example, “when the temperature rises over 70 degrees F send me an email” or “Whenever anyone mentions me in a tweet, have my device ring a bell” or even, “flash the subject line of any emails I receive to me as Morse Code.”

If This Then That

To get started with IFTTT, you will need to visit http://ifttt.com and register. Once you are registered you can get started writing your own recipes, or for that matter use recipes that other people have contributed.

IFTTT integrates lots of other web services such as Facebook, Twitter and Gmail and to be able to create and use recipes that use these web services, you will have to grant IFTTT access to these accounts by logging into these separate accounts as you set up a recipe that uses them.

At this point, you might like to explore IFTTT read their instructions on getting started and even try out a simple recipe or two before you start using it with the Photon/Core.

Project 11. Temperature eMail Alerts

This project uses the same code on the Photon/Core as Project 10. In fact it has exactly the same hardware as project 10 too.

So, if you have not already done so, build Project 10 and try it out before moving on to this project.

In this project, the webpage that we created to show the temperature in Project 10 will not be used (although you can still use it for testing). Instead, you are going to create an IFTTT recipe that monitors the temperature Variable and sends you an email when a temperature of 70 degrees F is exceeded.

Make Project 10 and power-up the Photon/Core so that IFTTT can find it and then start by creating a new IFTTT Recipe by clicking on the Create a Recipe button (Figure 6-1).

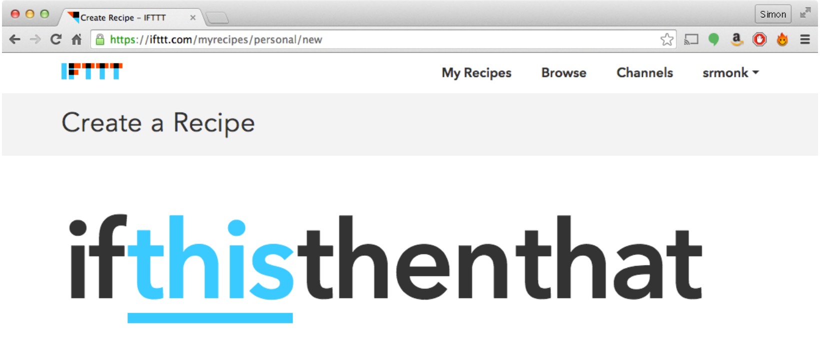

Create Recipe - IFTTT
https://ifttt.com/myrecipes/personal/new
Simon
IFTTT
My Recipes
Browse
Channels
srmonk
Create a Recipe
ifthisthenthat

Figure 6-1. *Figure 6-1. Creating a New Recipe in IFTTT*

The big clue here is the hyperlinked word “this” that is inviting you to click on it so that you can define a trigger. This will take you to a screen-full of icons for all the various web services that IFTTT knows about, that can act as a “trigger channel." Scroll down the list until you find the icon for Spark, shown in Figure 6-2. If this is the first time you have used Spark with IFTTT then you will be asked to log into Spark. Remember you are already logged into IFTTT, the login details you need to enter here are the username and password for your Spark.io account, so that IFTTT can have access to it.

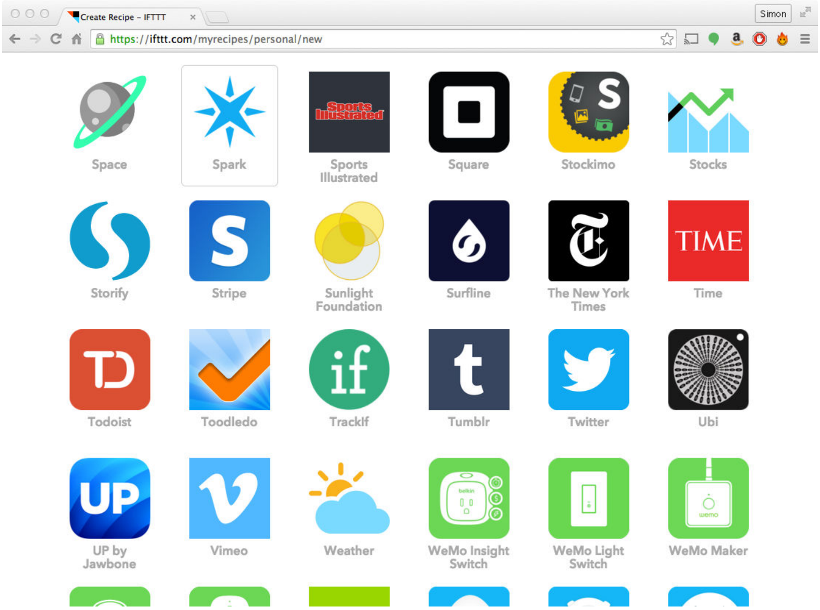
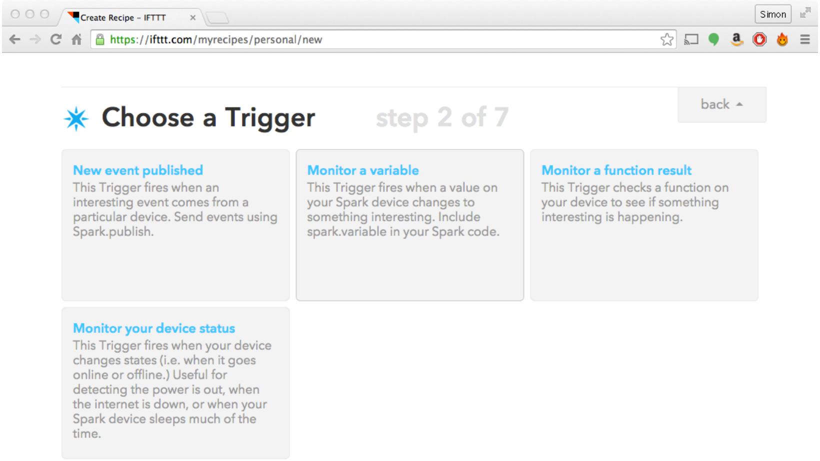

Create Recipe - IFTTT
Simon
https://ifttt.com/myrecipes/personal/new
Space
Spark
Sports Illustrated
Square
Stockimo
Stocks
Storify
Stripe
Sunlight Foundation
Surfline
The New York Times
Time
Todoist
Toodledo
TrackIf
Tumblr
Twitter
Ubi
UP by Jawbone
Vimeo
Weather
WeMo Insight Switch
WeMo Light Switch
WeMo Maker

Figure 6-2. *Figure 6-2. Selecting a Trigger Channel in IFTTT*

Click on the Spark Icon and you will be presented with a list of Triggers to chose from (Figure 6-3).



Choose a Trigger

New event published
This Trigger fires when an interesting event comes from a particular device. Send events using Spark.publish.

Monitor a variable
This Trigger fires when a value on your Spark device changes to something interesting. Include spark.variable in your Spark code.

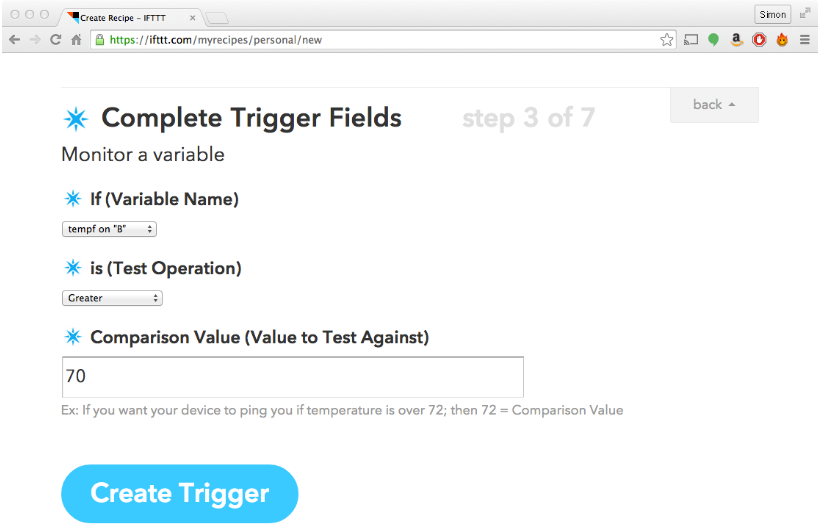
Monitor a function result
This Trigger checks a function on your device to see if something interesting is happening.

Monitor your device status
This Trigger fires when your device changes states (i.e. when it goes online or offline.) Useful for detecting the power is out, when the internet is down, or when your Spark device sleeps much of the time.

Figure 6-3. *Figure 6-3. Selecting a Trigger in IFTTT*

Select the option "Monitor a variable," since you want to be able to monitor the value of the tempF Variable on your Photon/Core.

This will open the simple form shown in Figure 6-4.



Complete Trigger Fields

Monitor a variable

If (Variable Name): tempf on "B"

is (Test Operation): Greater

Comparison Value (Value to Test Against): 70

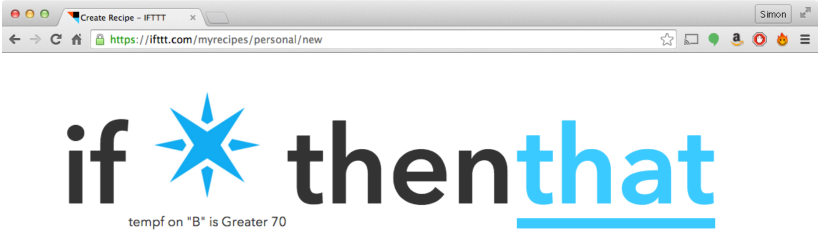
Ex: If you want your device to ping you if temperature is over 72; then 72 = Comparison Value

Figure 6-4. *Figure 6-4. Configuring the Trigger*

Here, when you click on the “If (Variable name)” field, you will be shown a drop-down list of all the Variables on all your Photons (and Cores). In Figure 6-4 you can see that “tempf on B” is selected (my Photon is rather unimaginatively called B).

In the next field you need to select the “Test Operation." We want to trigger something when the temperature exceeds 70, so select the Test Operation of “Greater." If you wanted to set up alerts for temperatures falling below some threshold (say to warn you of pipes about to freeze) then you could use “Less” here.

In the final field of Figure 6-4 set the “Comparison Value” to be 70 and then click on Create Trigger to move onto the next stage (Figure 6-5). If you live somewhere warm then you may want to increase 70 to perhaps 80, so that when testing the recipe, it isn’t triggered until you deliberately warm up the sensor.


Create Recipe - IFTTT
https://ifttt.com/myrecipes/personal/new
Simon
if
then
that

tempf on "B" is Greater 70

Figure 6-5. *Figure 6-5. Defining an Action in IFTTT*

Now you are being invited to complete the “That” part of the recipe. Notice that underneath the If part of the recipe it says “tempf on ‘B’ is Greater 70." Now click on the “That” hyperlink and you will be given a selection of Action Channels to select from (Figure 6-6).

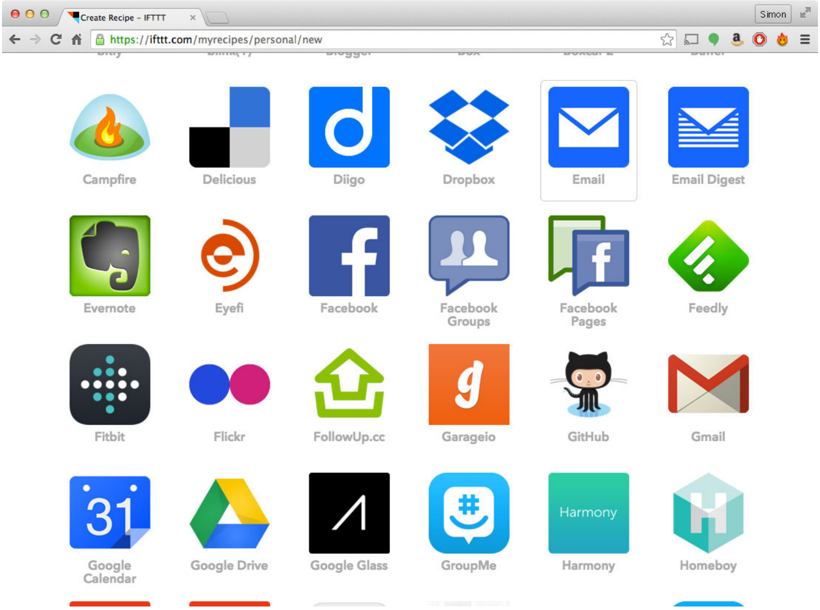
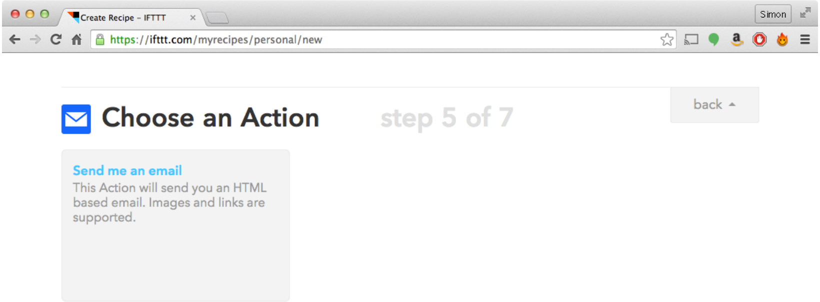

Create Recipe - IFTTT
Simon
https://ifttt.com/myrecipes/personal/new
Campfire
Delicious
Diigo
Dropbox
Email
Email Digest
Evernote
Eyefi
Facebook
Facebook Groups
Facebook Pages
Feedly
Fitbit
Flickr
FollowUp.cc
Garageio
GitHub
Gmail
Google Calendar
Google Drive
Google Glass
GroupMe
Harmony
Homeboy

Figure 6-6. *Figure 6-6. Selecting an Action Channel.*

The action we want is to have an email sent to us. The email will come from IFTTT's email server and they already know your email address from when you registered, so select the Email icon from the list of Action Channels.

Next, you need to select the action you want to take with that Channel and there is only one choice ("Send me an email") shown in Figure 6-7.

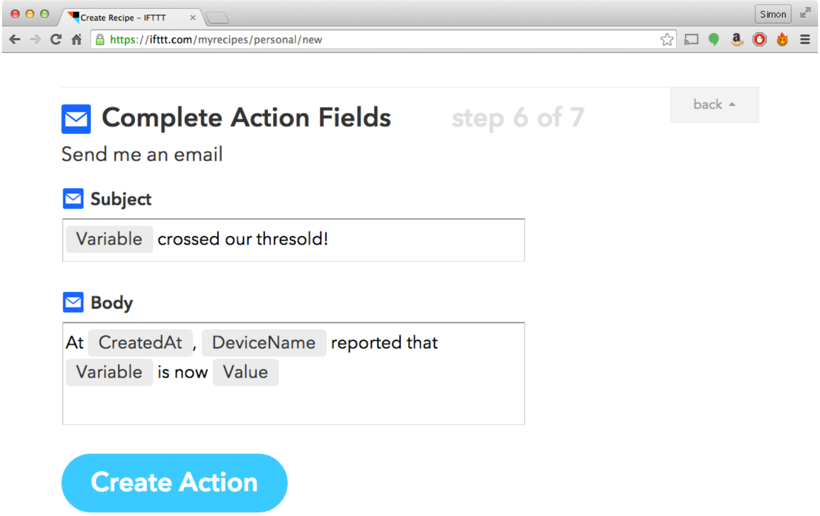

Create Recipe - IFTTT
Simon
https://ifttt.com/myrecipes/personal/new
back

Choose an Action step 5 of 7

Send me an email
This Action will send you an HTML based email. Images and links are supported.

Figure 6-7. *Figure 6-7. Selecting the Send Email Action*

Click on the “Send me an email” option and another form will appear for you to complete (Figure 6-8).



Complete Action Fields step 6 of 7

Send me an email

Subject: Variable crossed our thresold!

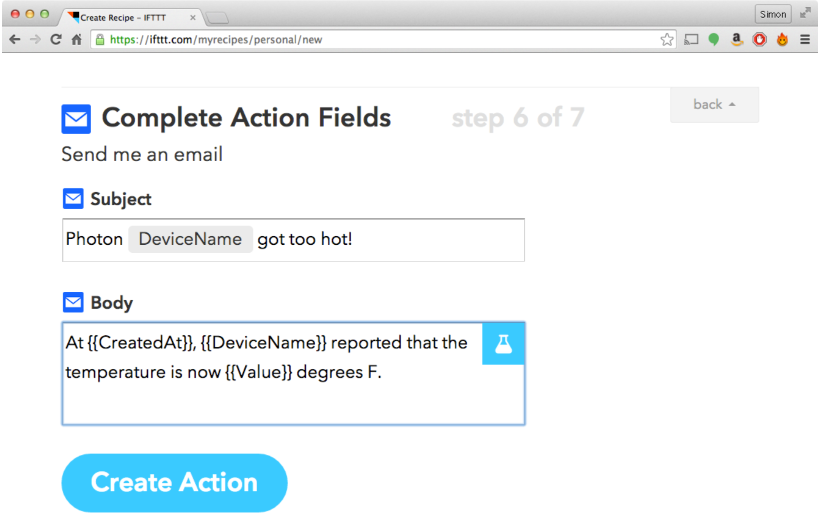
Body: At CreatedAt, DeviceName reported that Variable is now Value

Figure 6-8. *Figure 6-8. Options for Send Email.*

This form will determine what will go into the email that gets automatically sent. It is a mixture of regular text and also “ingredients” (Variable, CreatedAt, DeviceName and Value). These ingredients can be used in both the Subject and body of the email. Their meanings are as follows:

- Variable - the name of the Spark.io Variable on the Photon/Core, in this case, always “tempf”
- CreatedAt - the date and time when the email was triggered
- DeviceName - the name you gave to your Photon or Core
- Value - the actual value of the Variable at the time that the email was triggered

Change the fields so that they look like Figure 6-9. Note that when you go to edit a field, the variables in that field will be converted to text enclosed in to mark them out as variables.


Create Recipe - IFTTT
https://ifttt.com/myrecipes/personal/new
Simon
Complete Action Fields
step 6 of 7
back
Create Action

Send me an email

Subject

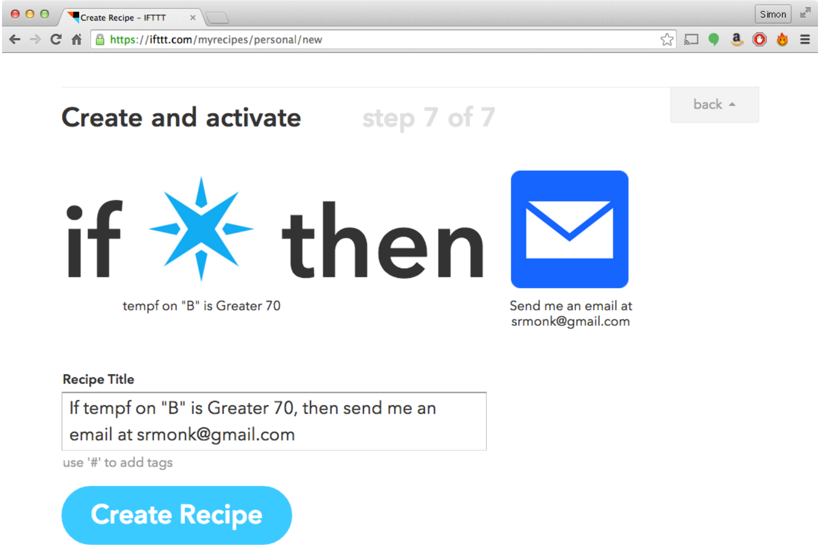
Photon DeviceName got too hot!

Body

At {{CreatedAt}}, {{DeviceName}} reported that the temperature is now {{Value}} degrees F.

Figure 6-9. *Figure 6-9. The Send Email Option Configured*

Click on the "Create Action" button and you will come to the final screen, which asks you to "Create and Activate" the recipe (Figure 6-10). Once you do this the recipe will go live! IFTTT only guarantees to check every 15 minutes, so you may not receive an email immediately and you are not going to be flooded with emails.



Create Recipe - IFTTT
Simon
https://ifttt.com/myrecipes/personal/new
Create and activate
step 7 of 7
back
if
then

tempf on "B" is Greater 70

Send me an email at srmonk@gmail.com

Recipe Title

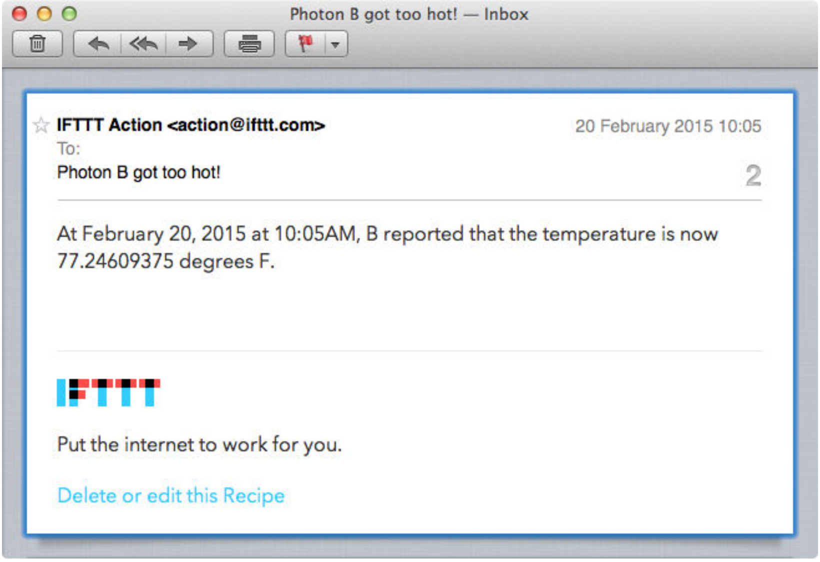
If tempf on "B" is Greater 70, then send me an email at srmonk@gmail.com

use '#' to add tags
Create Recipe

Figure 6-10. *Figure 6-10. Creating and Activating the New Recipe*

Also, you may need to warm up your sensor to over 70 degrees F by pinching it between your fingers, or leaning it against a warm mug of coffee (which you can drink while you wait for the email).

Eventually, you should get an email something the one in Figure 6-11.



IFTTT Action <action@ifttt.com>

20 February 2015 10:05

To:

Photon B got too hot!

At February 20, 2015 at 10:05AM, B reported that the temperature is now 77.24609375 degrees F.

Put the internet to work for you.

Delete or edit this Recipe

Figure 6-11. *Figure 6-11. An Email Notification from IFTTT*

Of course there are all sorts of other actions that you could trigger as a notification, so you may like to go back and modify the Action part of the recipe to, say, send a Tweet or Facebook status update.

In the next project you will learn how to use a Photon or Core with IFTTT to act as an action rather than a trigger.

Project 12. Ring a Bell for Tweets

This is another project that includes hardware and software that you already created in an earlier project. In this case, the Project in question is Project 7, where you used a Relay Shield and connected a bell to one of the relays, so that you could turn the bell on and off over the Internet.

In this case, you will use IFTTT to ring the bell every time someone mentions your name in a Tweet.

The bell used in Project 7 was kind of loud, and rather than have it sound for any longer than necessary, in this project, it will just be turned on for a fraction of a second. Just long enough for it to alert you, without driving you insane.

You could of course wire up a light-bulb or some other device to the relay if you want something a bit quieter.

Software

The program for the Photon/Core is similar to that of Project 7, so please refer back to that project for a full description of the code. You can find the code in the files p_12_tweet_bell in the PHOTON_BOOK library examples.

The program is still general purpose, in that it allows you to control any of the relays. However, the Function name has changed to "relaycontrol" and an extra option of specifying an "on" time has been added. This will allow us to make the duration of the bell sounding just short enough to alert us to the tweet.

The syntax of the command is backwards compatible with the old relay command, so that you can still use the command "31" to turn relay 3 on and "30" to turn relay 3 off again. However, now there is a different separate command "P" in place of "1" and "0" for on and off that then takes another parameter of the duration in milliseconds for the relay to be on. For example "3P-1000" will turn relay three on for one second.

The code for relaySwitcher function that implements this is listed below:

```
int relaySwitcher(String command) {
    // "11", "10", "1P-1000" - P for pulse duration in millis
    int relayNumber = command.charAt(0) - '0';
    char action = command.charAt(1);
    if (action == '1') {
        digitalWrite(relayPins[relayNumber-1], HIGH);
    }
    else if (action == '0') {
        digitalWrite(relayPins[relayNumber-1], LOW);
    }
    else if (action == 'P' || action == 'p') {
        int duration = command.substring(3).toInt();
        digitalWrite(relayPins[relayNumber-1], HIGH);
        delay(duration);
        digitalWrite(relayPins[relayNumber-1], LOW);
    }
    return 1;
}
```

The new part of the function (highlighted in the code) first checks to see if the action is “P” or “p” and then uses a substring from position 3 to the end of the command and converts this from a string to an integer using “toInt.”

Turning the relay on for the right duration is then just a matter of setting the appropriate relay pin high, delaying for the required number of milliseconds and then turning it off again.

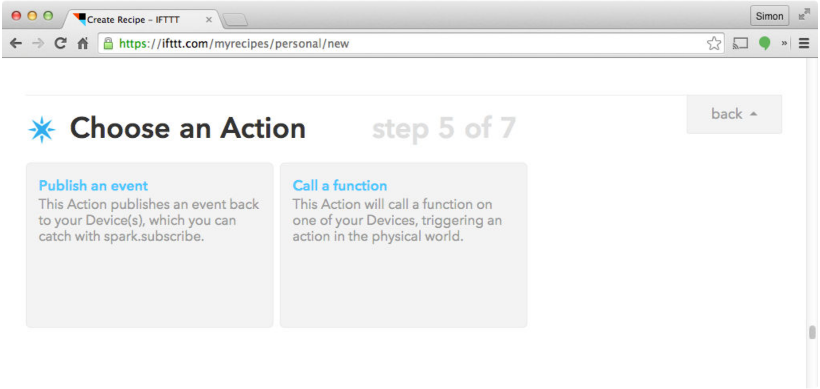
Flash the program onto your Photon/Core so that IFTTT will be aware of the new Action.

IFTTT

Login to your IFTTT account and then, when choosing a Trigger Channel, select Twitter. If its the first time that you have used Twitter from IFTTT then it will ask you to authenticate with Twitter, so that IFTTT can monitor your Twitter account. There are quite a few triggers that you can then select from within Twitter. The one I decided on was “New Mention of You.”

Select “New Mention of You” and the next screen will tell you that there are no fields to complete, so just click “Create Trigger." You can now move on to the “that” section of the recipe.

Click on the “Spark” icon in the Action channel source and you will be given the choice shown in Figure 6-12.



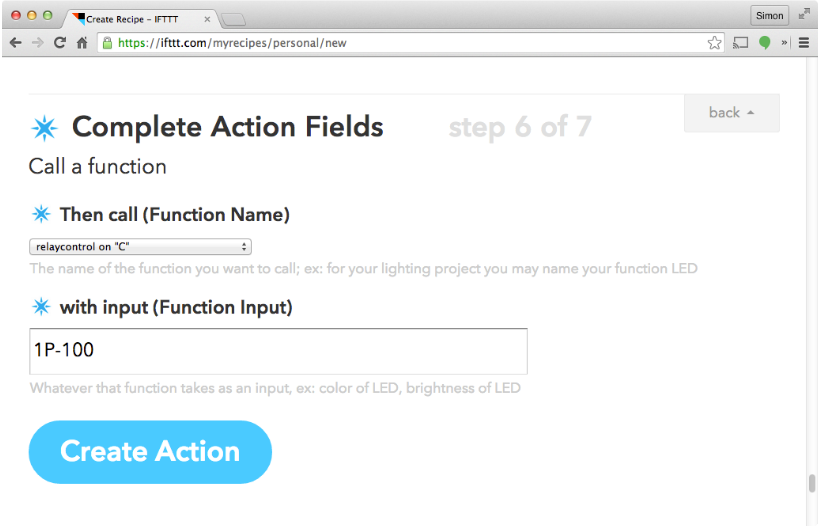
Choose an Action step 5 of 7

Publish an event
This Action publishes an event back to your Device(s), which you can catch with spark.subscribe.

Call a function
This Action will call a function on one of your Devices, triggering an action in the physical world.

Figure 6-12. *Figure 6-12. Selecting a Spark Action*

Select the option "Call a function" and you will be asked to "Complete Action Fields." From the "Then call" dropdown, select "relay control" on whatever Photon or Core you just flashed, and then delete the contents of the "with input" field with the text "1P-100" (see Figure 6-13).



Create Recipe – IFTTT

https://ifttt.com/myrecipes/personal/new

Complete Action Fields step 6 of 7

Call a function

Then call (Function Name)

relaycontrol on "C"

The name of the function you want to call; ex: for your lighting project you may name your function LED

with input (Function Input)

1P-100

Whatever that function takes as an input, ex: color of LED, brightness of LED

Figure 6-13. *Figure 6-13. Configuring a Spark Action*

Click Create Action and then click Create Recipe.

That's it, the code is all ready. You now just need to setup the hardware.

Hardware

The hardware for this project is just the same as that of Project 6 (see Figure 6-14). So, if you have not already made Project 6, you might like to go back to Chapter 5 and build it now.

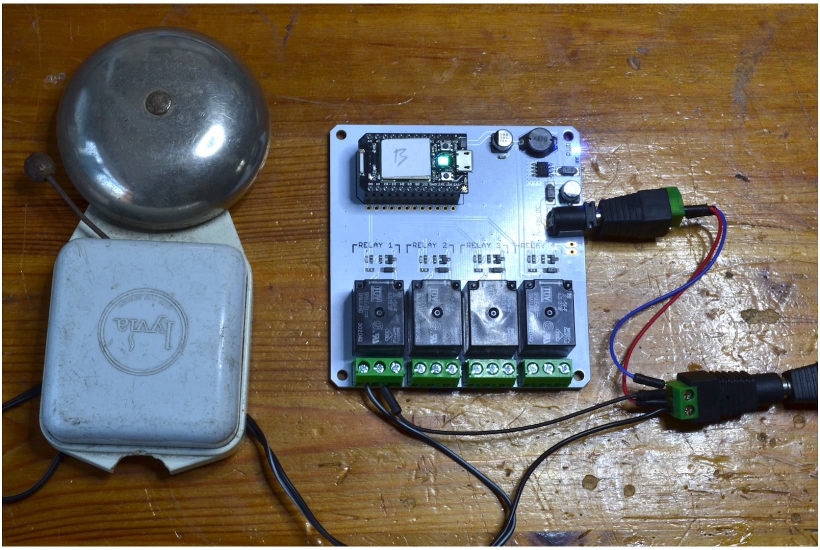


Figure 6-14. *Figure 6-14. The Hardware - from Project 6*

Using the Project

To test out the project, you need to mention yourself in a Tweet, or get someone to mention you. When IFTTT notices this, then the bell will ring. You can adjust the duration of the bell ring by editing the parameter to the Action in the IFTTT recipe.

Note that IFTTT only guarantees to check triggers every 15 minutes, so you will probably have to wait a minute or two before you know if the project worked.

As an alternative to alerting you of mentions would be to alert you of new followers or when you tweet using a specific Hashtag. Or, you could chose another type of trigger entirely, such as sending an email (use the "Email" trigger source) or simply at a certain time of day using the "Date & Time" trigger source.

Project 13. Flash Email as Morse Code

The final IFTTT example project takes the Morse Code flasher of Project 8 and gives it an IFTTT twist. The twist is that when you send an email to IFTTT the subject of that email will be flashed as Morse code.

Software

The code running on the Photon/Core is exactly the same as for Projects 8. Just flash your Photon with the program P_08_Morse_Function from the PHOTON_BOOK library.

Hardware

The hardware for the project is also exactly the same as for Project 8. The software will actually flash the built-in LED on D7, so if you do not want to use an external LED or buzzer, then you can just use the Photon/Core on its own without any components attached to it at all.

If you have decided that you want a buzzer and external LED then follow the instructions for Project 8 in Chapter 6.

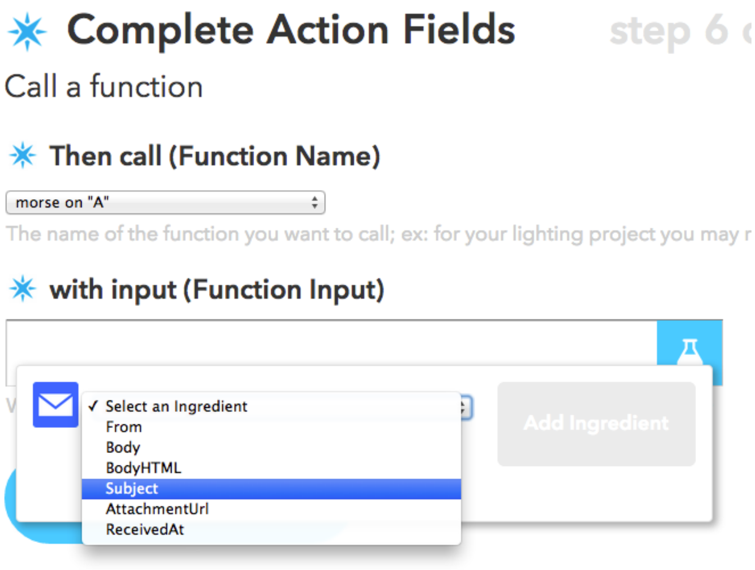
IFTTT

To create the IFTTT recipe for this project, create an new Recipe and then select Email as the trigger source. In IFTTT the Email trigger source means emails to and from the IFTTT email server. In this case, select the trigger “Send IFTTT any email." There are no fields to enter, so at the next screen click on Create Trigger.

The Action is more interesting. Select Spark as the Action Channel and then the “Call a Function” action.

In the “Then call (Function Name)” field select the Function “morse” on whatever Photon/Core you have just flashed the program onto.

Now you need a way of passing the subject of the email to the “morse” Function. To do this kind of thing, IFTTT provides the concept of an “ingredient” for the recipe. Delete the contents of the “with input (Function Input)” field and click on the beaker icon on the right. The popup-list in Figure 6-15 will then appear and you can click on the “Subject” ingredient.


Complete Action Fields

Call a function

Then call (Function Name)

morse on "A"

The name of the function you want to call; ex: for your lighting project you may r

with input (Function Input)

Figure 6-15. *Figure 6-15. Selecting Ingredients for your Recipe*

This will insert the text into the field. Click on Create Action and then Create Recipe.

Using the Project

To test out the project, send an email to trigger@recipe.ifttt.com with the subject line of the text that you want to be flashed and beeped as Morse Code. This email must be sent from the email account that you used to register with IFTTT.

Summary

In the next chapter, you will learn how to use a Photon/Core to create a small roving robot that is controlled from a webpage and that also displays its battery voltage and the distance to any obstacle in front of it.

7/Robotics

The Photon's small size and wireless connectivity makes it a great choice for robotics projects. In this chapter, you will learn how to use a Photon or Core with a Photon Bot shield to control a small roving robot.

The robot is also fitted with an ultrasonic range finder module to help prevent it from bumping into things.

Project 14. Web Controlled Robot

This project makes use of a robot controller board called the Photon Bot by MonkMakes. You may guess by the company name that I had something to do with the design of this product, which you can buy from Spark and other sources, see Appendix A.

The board is combined with a chassis kit that provides a base and gearmotors to drive the robot, as well as wheels and a battery pack.

Figure 7-1 shows the finished rover and Figure 7-2 the web page to control it and provide feedback about the state of the rover's battery and the distance to any obstacles in front of it.

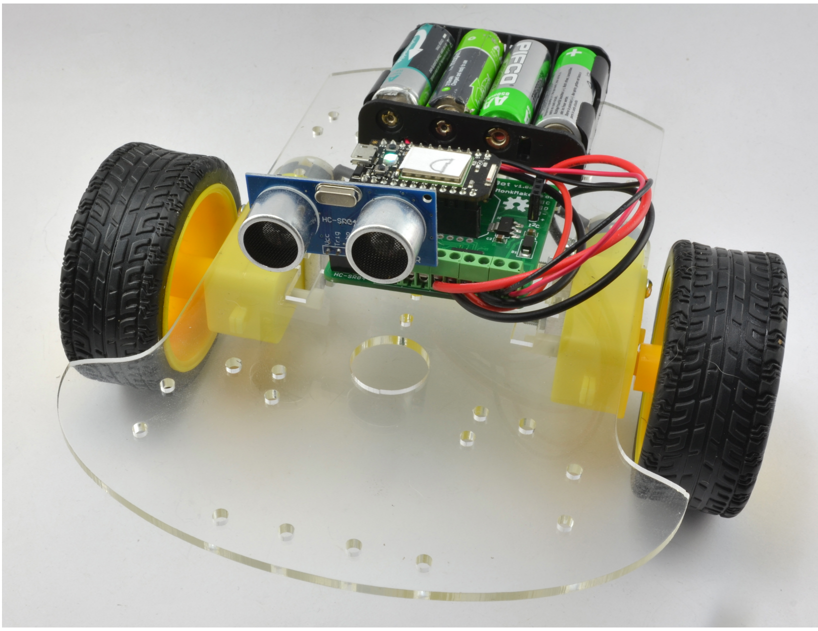


Figure 7-1. *Figure 7-1. A Photon Roving Robot*

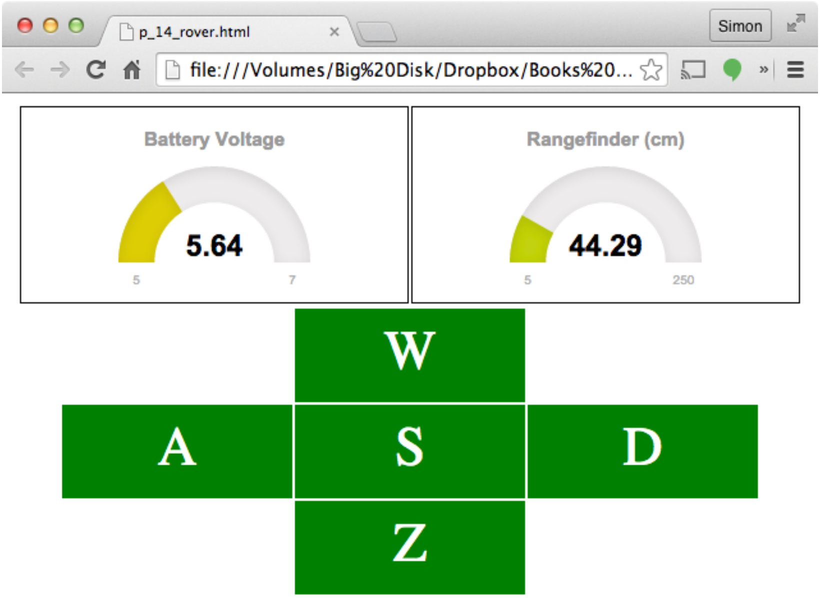

p_14_rover.html
Simon
file:///Volumes/Big%20Disk/Dropbox/Books%20...
Battery Voltage
5.64
5
7
Rangefinder (cm)
44.29
5
250
W
A
S
D
Z

Figure 7-2. *Figure 7-2. A Webpage to Control the Rover*

Parts

To build this project, you are going to need the following parts in addition to your Photon/Core.

Table 7-1. *Parts Bin*

Part	Description	Appendix A code
Photon Bot	Photon Bot Robot Controller Shield	M2
Rangefinder	HC-SR04 ultrasonic rangefinder	M3
Chassis Kit	Rover robot chassis kit (6V)	H6
Hookup wire	Wire to connect the motors to the Photon Bot	H3

There are many low-cost robot chassis on the market and most will work with the Photon Bot. Look for a kit with 6V motors, and a 4 x AA battery holder, although lower voltage motors will also work.

You will only need the hookup wire if the gearmotors supplied with the chassis kit do not have wires attached. In which case, you will need to solder wires about 6 inches long to each motor terminal.

Software (Photon)

It's a good idea to program your Photon/Core with the app for this project before attaching it to motors, where some inadvertent use of output pins on a previous project could have your rover marching off the end of your table.

The program for this project is the example program provided in the Photon Bot library that makes the Photon Bot easy to use.

Find the PHOTON_BOT library, by typing "PHOTON_BOT" into the search field below "Community Libraries" in the Web IDE.

When you do this, you will see the library files. Select the file WEBROVER.CPP and then click on the button "USE THIS EXAMPLE." This will create a copy of the example program that we can use to program the Photon/Core, but can also be modified if we want to change things a little.

Before the WEBROVER example can be flashed onto a Photon or Core, you will need to find a couple of libraries that are used by the PHOTON_BOT library and also the HC_SR04 library needed for the rangefinder. To import each of these libraries for use with the example WEBROVER you must find each library (HC_SR04, SPARKINTERVALTIMER and SOFTPWM) in turn in the Community Libraries area and then click on the INCLUDE IN APP button, then when the Web IDE prompts with "Which app," select WEBROVER from the list and click the ADD TO THIS APP button. Repeat this for all three libraries. You will notice that each time that you do this a new #include statement will appear at the top of the program.

The libraries take care of most of the code for the rover for us. Here is the listing for the WEBROVER.

```
#include "HC_SR04/HC_SR04.h"
#include "SparkIntervalTimer/SparkIntervalTimer.h"
#include "SoftPWM/SoftPWM.h"
#include "Photon_Bot/Photon_Bot.h"
```

```
double volts = 0.0;
double distance = 0.0;

PhotonBot p = PhotonBot();
HC_SR04 rangefinder = HC_SR04(p.trigPin, p.echoPin);

void setup() {
    Spark.function("control", control);
    Spark.variable("volts", &volts, DOUBLE);
    Spark.variable("distance", &distance, DOUBLE);
}

void loop() {
    volts = p.batteryVolts();
    distance = rangefinder.getDistanceCM();
}

int control(String command) {
    return p.control(command);
}
```

Note that the extra comments added next to each include that appear automatically after using a library have been removed for clarity.

This program uses two Variables: volts, which contains the battery voltage, and distance, which contains the last rangefinder measurement in cm.

After the two double variables, the PhotonBot and rangefinder are initialized and assigned to the variables *p* and *rangefinder* respectively. When the rangefinder is initialized, the pins for it to use are taken from the photon bot's library as *p.trigPin* and *p.echoPin* (see sidebar on Ultrasonic Rangefinders).

Ultrasonic Rangefinders

Ultrasonic rangefinders work by timing how long it takes for a pulse of ultrasound to bounce off an obstacle and return to the sensor. Using the speed of sound, the distance is calculated.

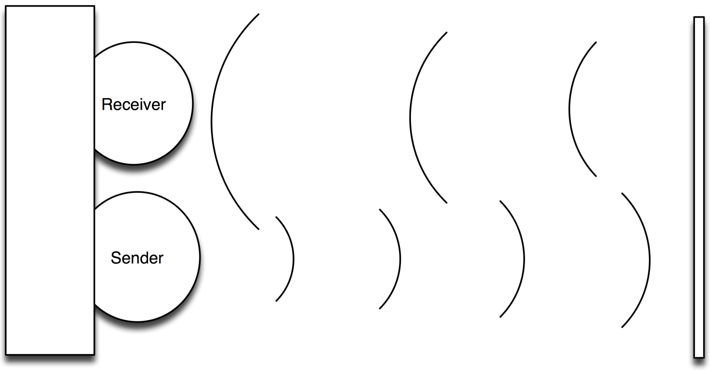


Figure 7-3. *Figure 7-3. How an Ultrasonic Rangefinder Works*

The HC-SR04 module used for this has power pins and two additional pins, trigger and echo. When the trigger pin is taken high for a fraction of a second, the module emits a pulse of ultrasound. When that pulse returns, the echo pin will indicate its return.

The *setup* function defines a Function called *control*. The *control* function uses high level commands like “F-100”. The first letter is the direction that the bot is to travel in (F-Forwards, B-Backwards, L-turn left, R-turn right) and the parameter after the - is the speed where 100 is full speed, 50 would be half speed, etc. The command S without any speed parameter stops the bot.

The two Variables are also defined in setup: *volts* and *distance*.

The *loop* function just updates both variables with the current battery voltage and distance to an obstacle.

The function *control* is the handler for the *control* Function and just passes on whatever command it is given to the function of the same name within the Photon Bot library.

One tweak to mention here is that if your chassis has motors of a lower voltage than that of the battery pack (say 3V) then you can change the code slightly to compensate for this. Replace the line of code below:

```
PhotonBot p = PhotonBot();
```

with:

```
PhotonBot p = PhotonBot(6.0, 3.0);
```

The first parameter (6.0) is the battery voltage. This does not need to be exact, if you have 4 x AA cells, set this to 6.0. The second parameter is the motor voltage, so set this to 3V if you have 3V motors. If you are not sure about the voltage of the motors, start by setting them to 3V. You can always increase this if it makes the motors run too slowly.

You can flash the WEBROVER app onto your Photon/Core while it is just connected to USB before you start attaching it to the hardware and its battery pack.

Software (Webpage)

The webpage for this project uses a combination of techniques that you have already seen in earlier projects and is too long to list here, so you might want to open the file p_14_Rover.html in a text editor.

This project actually uses three URLs, for the two Variables and the control Function.

```
var accessToken = "cb8b348000e9d0ea9e354990bbd39ccbfb57b30e";
var deviceID = "55ff6b065075555322151487"
var volts_url = "https://api.spark.io/v1/devices/"
                + deviceID + "/volts";
var distance_url = "https://api.spark.io/v1/devices/"
                + deviceID + "/distance";
var control_url = "https://api.spark.io/v1/devices/"
                + deviceID + "/control";
```

Remember that you will need to change the *accessToken* and *deviceID* for your account details.

Since there are now two Variables to be displayed in gauges, there are two corresponding callback functions to update the gauges, *callbackVolts* and *callbackDistance*. Both are much the same as earlier examples. The callback for the battery voltage finishes by scheduling another reading after 10 seconds, rather than the 1 second update of the rangefinder reading, as the battery voltage shouldn't change that quickly.

The web page (see Figure 7-2) has buttons to control direction, but also maps key-presses on the keyboard to the various direction commands. The following line is responsible for intercepting key presses while you are on this webpage and calls the function *keyPress* every time a key is pressed.

```
$(document).keypress(keyPress);
```

The keyPress function is listed below:

```
function keyPress(event){
    code = event.keyCode;
    if (code == 119) {
        sendControl('F-100');
    }
    else if (code == 97) {
        sendControl('L-50');
    }
    else if (code == 115) {
        sendControl('S');
    }
    else if (code == 100) {
        sendControl('R-50');
    }
    else if (code == 122) {
        sendControl('B-75');
    }
}
```

Each key has a character code, based on the ASCII code for that letter. If you search the Internet you will find information about ASCII codes for letters. In this case, *w* is 119, *a* is 97, *s* is 115, *d* is 100 and *z* is 122. When a particular key is pressed the relevant command is sent to the Photon/Core.

One feature of the code for the volts gauge is that the gauge should go red as the battery becomes exhausted. This is achieved using a color list associated with the gauge. The three colors being red, orange and green as # hex strings.

```
var volts_gauge = new JustGage({
    id: "voltsGauge",
    value: 0,
    min: 4,
    max: 7,
    levelColors: ["#FF0000", "#FFFF00", "#00FF00"],
        title: "Battery Voltage"
    });
```

Hardware

Before you start attaching any electronics, you will need to assemble the robot chassis. Most kits will comprise the following items:

- A laser-cut acrylic plastic base
- Two gearmotors (ideally 6V)
- Two wheels to fit the gearmotors
- A battery box to fit from 4 to 6 AA battery cells
- A castor or universal wheel to be fitted at one end of the chassis

Before you assemble the chassis, you may need to solder leads to the gearmotors that are long enough to reach around to the screw terminals of the Photon Bot board.

Once the chassis is assembled, work out where you want to place everything on the chassis. The heaviest part of the rover will be the battery holder (once it has batteries in it) so make sure that this is positioned inside the triangle formed by the castor and the two drive wheels otherwise the rover might tip over.

Remember that the rangefinder will need a clear view in front of it to be able to detect obstacles. If you are lucky, screw holes will be in the right place and you can screw the Photon Bot shield and battery box to the chassis base. If not, then self-adhesive Velcro (TM) pads are a great way of attaching things.

Figure 7-4 shows the two view of the wired-up rover. The rangefinder has been temporarily removed for a clearer view of the screw terminals.

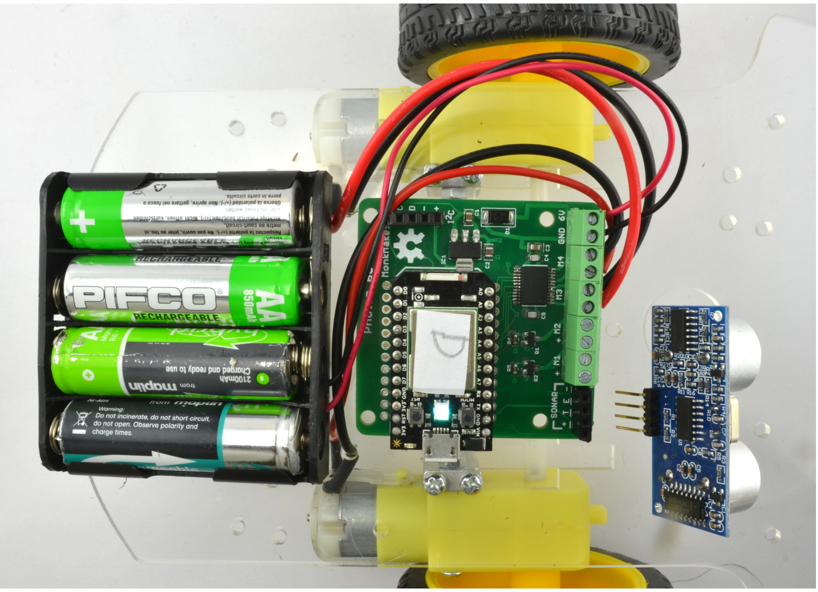


Figure 7-4. *Figure 7-4. Wiring the Photon Bot*

The battery box should have one red (+) and one black (-) lead. The red lead should go to the screw terminal labelled 6V and the black lead from the battery box is connected to the screw terminal marked GND.

The Photon Bot shield can actually control up to four motors, but only the pairs of screw terminals marked M3 and M4 are capable of bi-directional operation, that is making the motor run both clockwise and counterclockwise, so we will use these rather than M1 and M2.

Thread the leads from the motors through a convenient hole in the chassis base and connect the left-hand motor leads to the pair of screw terminals marked M4 and the right-hand motor leads to the pair marked M3.

You will not really know if these leads for a particular motor are the right way around until you try and drive them. Reversing the leads on a DC motor causes it to rotate in the opposite direction.

Now you can plug the Photon/Core and Rangefinder onto the Photon Bot shield. Make sure that they are both the right way around. You can also fit the batteries into the battery holder now.

If the battery holder does not have a switch, then pull one end of one of the batteries out of the battery box as a crude switch.

Using the Project

Chances are you are going to need to swap over a few of the motor wires, so its best to get all of that sorted out while the wheels are still off the motors. That way, you can sit the rover in front of you without worrying about it wandering off.

Open p_14_rover.html in your web browser and press the F button (or key) and watch what the motor shafts do for moving forward. One of the following situations will apply.

- They are both moving correctly - hurray you are finished!
- One of the motors is rotating in the wrong direction. If so, swap over the wires for that motor only, or both motors if both are wrong.

Now press A or the a key on your keyboard and watch what the motors do. If you performed the step above correctly, then the motors will be moving in opposite directions. The question is, whether this is turning you to the left (counterclockwise) or right (clockwise). If the answer is counter clockwise, then all is well and you are done, otherwise, you will need to swap the motor that is connected to M3 to be connected to M4 and vice versa. When doing this, keep the leads for each motor in the same order (for example red on the left, black on the right).

The rover is fairly power hungry, so when its not in use, pull one end of one of the batteries out to turn it off.

If you are using rechargeable batteries, then these are lower voltage than single-use AA batteries (generally 1.2V rather than 1.5V) and if the battery voltage falls too low, the Photon/Core will crash and you will loose control of the motors. Using a 5 or 6 cell battery holder will give you more drive time between charges if you are using rechargeable batteries. For single-use batteries, 4 x AA batteries should give you an couple of hours of driving time.

A nice refinement to the code might be to monitor the battery voltage and automatically stop the rover when the batteries get low.

You will find a number of other examples in the Library, you might like to try some of them out on you Rover.

Summary

In Chapter 8 you will learn about Spark's powerful publish and subscribe model and how to use it when making two or more Photons or Cores communicate with each other.

8/Machine to Machine Communication

Most of the projects in this book rely on the Photon or Core either being completely independent (once programmed) or being controlled through a web browser or other user interface over the Internet.

In this chapter, you will learn how to make one Photon/Core speak to another, or even many other Photons or Cores.

Publish and Subscribe

In the previous chapter, when you used IFTTT to monitor the temperature on a Photon/Core, IFTTT only checked the temperature Variable on the Photon every 15 minutes. The reason they limit it in this way is that otherwise there could be potentially huge volumes of web and email traffic if everyone's triggers were being monitored continuously. This type of checking on values is called "polling" and is not very efficient.

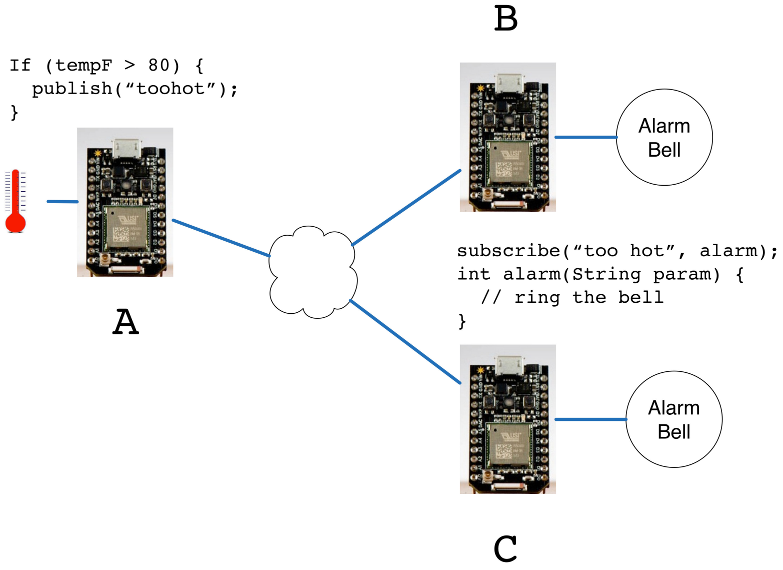
The alternative to polling in this situation would be to move the temperature testing to the Photon/Core, so that only when the level is exceeded would the device initiate communication by "publishing" a temperature-exceeded event to any services such as IFTTT or other Photons that are "subscribing" to that event.

In other words, a device or web service can subscribe to an event. That is, they can register an interest in that event, so that when that event is published, they are notified. This is a lot more efficient than continually polling and means that you don't have to (in the case of IFTTT) wait for 15 minutes before you receive your notification.

Temperature Monitor Example

If you have made projects 10 and 12, and have two Photons, then you can try out this project using the exact same hardware as those two projects. You will just need to flash the new software onto the Photons.

Figure 8-1 shows how the publish and subscribe model can be used to create a network of Photons all communicating with each other.



```
If (tempF > 80) {
  publish(“toohot”);
}
```

```
subscribe(“too hot”, alarm);
int alarm(String param) {
  // ring the bell
}
```

Figure 8-1. *Figure 8-1. Publish and Subscribe*

In this case, there is one Photon acting as a thermometer (A) and two Photons with alarm bells attached (B and C). The intention is that these alarm bells will ring whenever Photon A publishes a “toohot” event. Previously (in their setup functions) Photons B and C will have registered an interest in the “toohot” notification by subscribing to “toohot." Once they do this, then the Spark cloud service knows that every time Photon A says its "toohot" Photons B and C need to be informed.

The code on Photon A will look something like the code below. This code is a little flawed. You will fix this later, so for now, don't flash this onto your project 10 hardware, wait until it has been improved to solve excessive publishing.

```
#include "spark-dallas-temperature/spark-dallas-temperature.h"
#include "OneWire/OneWire.h"

int tempSensorPin = D2;
OneWire oneWire(tempSensorPin);
DallasTemperature sensors(&oneWire);

void setup() {
    sensors.begin();
}

void loop() {
  sensors.requestTemperatures();
  float tempC = sensors.getTempCByIndex(0);
  float tempF = tempC * 9.0 / 5.0 + 32.0;
  if (tempF > 80.0) {
      Spark.publish("toohot");
  }
}
```

Most of this code is concerned with measuring the temperature, so refer back to project 10 for more information on that. The key line is highlighted and shows just how easy it is to publish an event.

The code for Photon B (the relay and bell hardware) is shown below.

```
int relayPin = D0;
void setup() {
    pinMode(relayPin, OUTPUT);
    Spark.subscribe("toohot", soundAlarm);
}

void loop() {
}

void soundAlarm(const char *event, const char *data) {
    digitalWrite(relayPin, HIGH);
```

```
        delay(200);
        digitalWrite(relayPin, LOW);
    }
```

The *setup* function contains the code to subscribe to the "too-hot" event and specifies that the function *soundAlarm* should be called if such an event happens. You will notice that *soundAlarm* has some strange parameters, you will learn how these work later.

All though this pair of programs will work, there is (as I mentioned earlier) a slight flaw in the first program, and that is that the temperature is checked in the loop function and just so long as the temperature is over 80, the event will be published every time around the loop. So Photon A will be publishing the "too-hot" event over and over again, as fast as it can while the temperature is over 80. This is not very socially responsible programming. It would be better, if we used two events. One event that would be published when the temperature exceeded 80 and a second event ("tempnormal") that was published when the temperature had fallen back below say 78. The reason for a second threshold temperature of 78 rather than 80 is that if both threshold temperatures were 80, then at the point where the actual temperature was just around 80, the readings could alternate between 79 and 80 causing a flurry of both events.

The modified code for the thermometer end (Photon A) is listed below:

```
#include "spark-dallas-temperature/spark-dallas-temperature.h"
#include "OneWire/OneWire.h"

int tempSensorPin = D2;
OneWire oneWire(tempSensorPin);
DallasTemperature sensors(&oneWire);

boolean toohot = false;

void setup() {
    sensors.begin();
}

void loop() {
  sensors.requestTemperatures();
  float tempC = sensors.getTempCByIndex(0);
  float tempF = tempC * 9.0 / 5.0 + 32.0;
  if (tempF > 80.0 && toohot == false) {
      Spark.publish("toohot");
      toohot = true;
  }
  if (tempF < 78.0 && toohot == true) {
      Spark.publish("tempnormal");
      toohot = false;
  }
}
```

A new boolean variable *toohot* has been added to the program, so now, after reading the temperature, the “toohot” event is only published if the variable *toohot* is *false*, but the measured temperature has risen above 80. The variable *toohot* is then set to *true*, to prevent any further publishing of “toohot” until the temperature has fallen back below 78.

The second *if* statement takes care of this. If the measured temperature is below 78 and the *toohot* variable is currently *true*, then the “tempnormal” event is published and *toohot* set to *false*.

The code for the relay end of the project (Proton B) also needs to be modified to handle the new event (“tempnormal”).

```
int relayPin = D0;
void setup() {
    pinMode(relayPin, OUTPUT);
    Spark.subscribe("toohot", soundAlarm);
    Spark.subscribe("tempnormal", cancelAlarm);
}

void loop() {
}

void soundAlarm(const char *event, const char *data) {
    digitalWrite(relayPin, HIGH);
}

void cancelAlarm(const char *event, const char *data) {
    digitalWrite(relayPin, LOW);
}
```

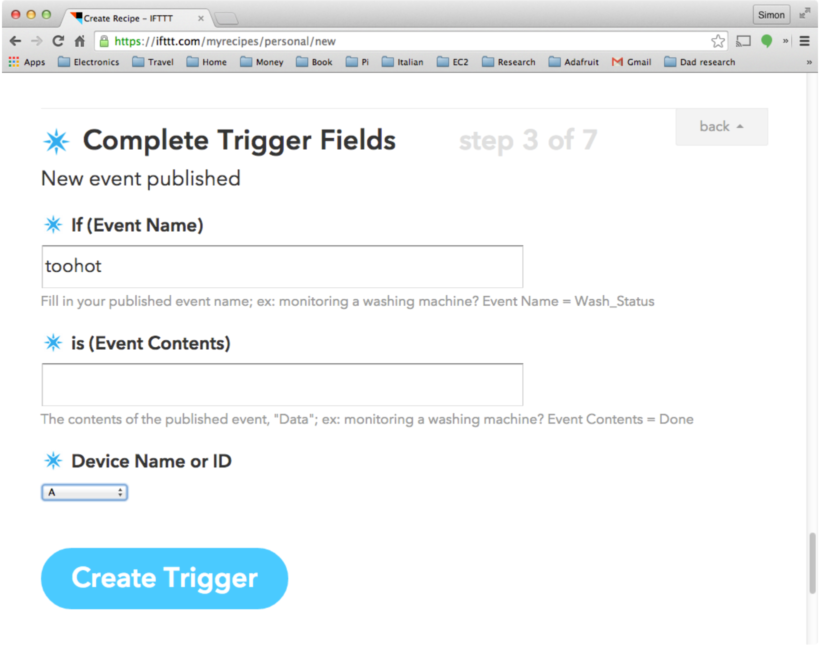
Now, setup subscribes to both “toohot” and “tempnormal” events. The “toohot” event will turn the alarm bell on and “tempnormal” will turn it off again -- just what we want.

If you want to try out these programs, the thermometer program is in the file ch_08_Temp_monitor_Pub and the relay program in ch_08_Temp_monitor_Sub. These will run on the hardware for Projects 10 and 12 respectively.

IFTTT and Publish/Subscribe

IFTTT is able to use the Spark Publish and Subscribe system. For example, you could arrange for an email to be sent when the temperature is exceeded not by monitoring a Variable as we did in Project 10, but by having an IFTTT recipe that subscribes to the “toohot” event would be much more efficient.

Create a new Recipe in IFTTT and then select Spark as the Trigger Source and instead of selecting “Monitor a Variable," select “New event published” as the trigger. Figure 8-2 shows the “Complete Trigger Fields” page.


Create Recipe – IFTTT
Simon
https://ifttt.com/myrecipes/personal/new
Apps
Electronics
Travel
Home
Money
Book
Pi
Italian
EC2
Research
Adafruit
Gmail
Dad research
back

Complete Trigger Fields step 3 of 7

New event published

If (Event Name)

toohot

Fill in your published event name; ex: monitoring a washing machine? Event Name = Wash_Status

is (Event Contents)

The contents of the published event, "Data"; ex: monitoring a washing machine? Event Contents = Done

Device Name or ID

A

Create Trigger

Figure 8-2. *Figure 8-2. Setting and Event Trigger*

Specify “toohot” as the event name and leave the Event Contents field blank. Finally, select the Photon/Core that is going to be sending the events.

Select an Action channel of Email and then the Action of “Send me an Email," as you did in Project 11.

When the recipe is finished and active, you will receive just one email each time the event is triggered. Pinch your temperature sensor between your fingers to warm it up and test out this example.

Advanced Publish and Subscribe

The example above uses publish and subscribe at its most simple and for many applications that will be all that you need. However the Spark framework for publish and subscribe is actually capable of more advanced features.

Spark.publish

You can read the full documentation for the publish command here: ch_08_Temp_monitor_Pub.

In the example above, the only parameter that we supplied to publish was the name of the event. In fact, you can supply the following additional parameters:

- data - This string could contain a a value to accompany the event. For example, the actual temperature could be sent, but this would need to be converted to a string first.
- time to live - This value specifies the number of seconds that the event should be allowed to live before it is automatically removed. This prevents too many events accumulating in the system. At the time of writing, this parameter is ignored by the Spark Cloud, which discards events after the default of one minute.
- public/private - The default is public, meaning that anyone can subscribe to these events. This allows for the possibility of interesting collaborative projects.

Spark.subscribe

The subscribe command also has an extra parameter, that follows the name of the handler function. This specifies the scope of the events that are to be subscribed to. You can use this third parameter to limit subscriptions to events coming from a particular Photon or Core by supplying the device's ID like this:

```
Spark.subscribe("toohot", soundAlarm,
                          "55ff70064955344339432587");
```

You can also limit the subscription to just your own devices using the command below:

```
Spark.subscribe("toohot", soundAlarm, MY_DEVICES);
```

Subscription name matching is actually even more subtle than this. It will match against however much of the name you supply, so the line below will still receive the "toohot" events.

```
Spark.subscribe("too", soundAlarm);
```

This opens up the possibility of organizing events into related families based on the event name. Note that the maximum size of an event name is 64 characters.

Project 15. Magic Rope

This project was inspired by a video made by Leena Ventä-Olkkonen, Tobi Stockinger, Claudia Zuniga and Graham Dean that showed how a public installation could be made that would allow large maps of the world to be positioned in various public spaces in a city. These maps would have short lengths of rope sticking out of holes on the map corresponding to other cities around the world. The idea is then that the public at any one of these cities (let's say London) could walk up to a rope at their map, pull on it and the paired rope in the other city (say New York) would be pulled into the map, attracting the attention of people near the installation. A gentle exchange of rope pulling could then occur across the world. You can view the original video here: https://www.youtube.com/watch?v=RzW9PJEpIhw

The original project was only ever developed as a concept and not actually implemented as a real installation. In this project, you will make a pair of "entangled" ropes that could be positioned in different cities. This could be used as a nice way of staying in touch with distant relatives.

Figure 8-3 and 8-4 shows the pair of Photons with their tempting lengths of string to be pulled. One is in a wooden box, adding to the mystery.

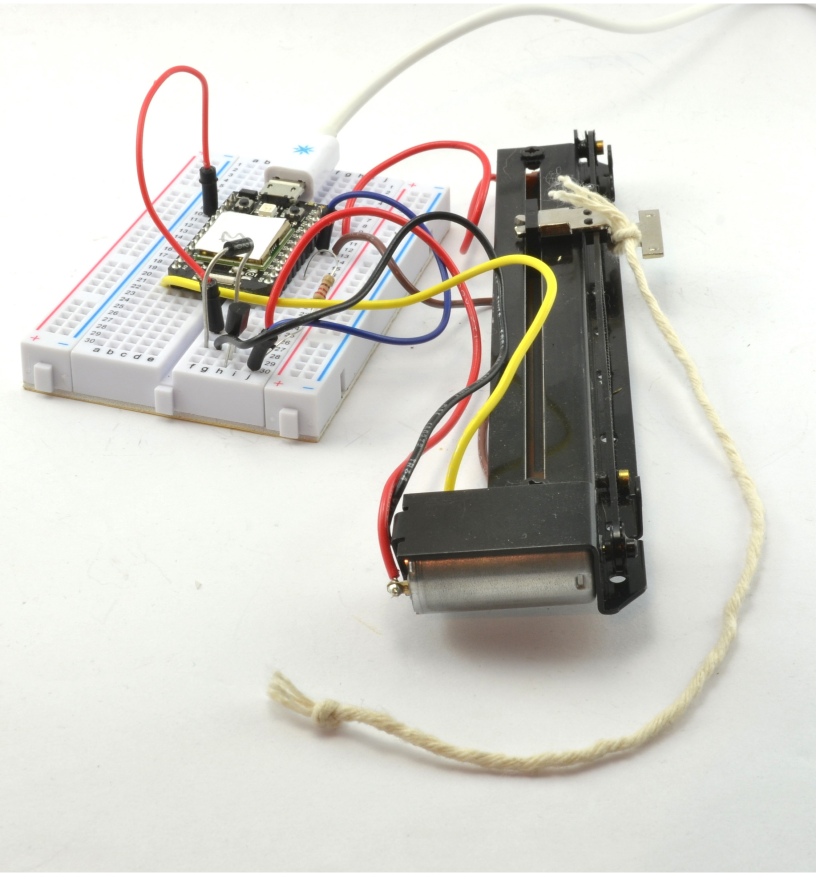


Figure 8-3. *Figure 8-3. The Magic Rope Project Naked*

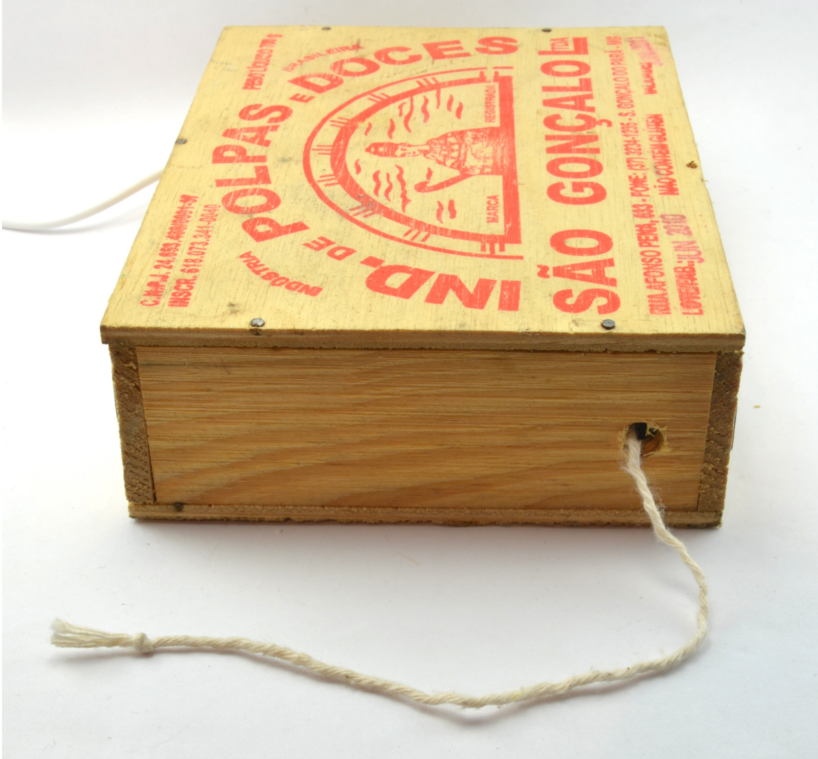


Figure 8-4. *Figure 8-4. The Magic Rope Project in a Box*

Parts

To build this project, you are going to need two sets of each of the following parts in addition to two Photons/Cores.

Table 8-1. *Parts Bin*

Part	Description	Appendix A code
R1	Slide Pot - Motorized	M4
R2	220Ω resistor	C1
Q1	2N3904 transistor	C10
D1	1N4001 diode	C11
	Male-to-male jumper wire	H4

	Half-sized breadboard	H5
	Assorted hookup wire	H3

These components are all included in the Maker's Kit apart from the motorized slide pots.

The sliding pots (potentiometers) used in this project are variable resistors intended for use in automated music mixing desks. You can adjust the resistance by sliding the "slider" up and down the length of the pot, but there is also a little motor that will move the slider using a toothed belt drive.

These motorized pots do not have leads attached to the pins, so this is one project where you will need to use a soldering iron and attach some leads to the pins.

Software

Both ends of this project have exactly the same software running on them and you can find it in the files P_15_Magic_Rope in the PHOTON_BOOK library.

```
int motorPin = D4;
int potPin = A0;

String thisID = Spark.deviceID();
boolean myTurn = true;
int maxPosn = 4000;
int minPosn = 3000;

void setup() {
    Spark.subscribe("pulled", remoteRopePulled);
    pinMode(motorPin, OUTPUT);
    moveSliderTo(maxPosn);
}

void loop() {
    int newLocalPosition = analogRead(potPin);
    if (newLocalPosition <  minPosn && myTurn) {
        Spark.publish("pulled", thisID);
        myTurn = false;
    }
}
```

```
void remoteRopePulled(const char *event, const char *data)
{
    String dataS = String(data);
    // ignore messages from yourself
    if (dataS.indexOf(thisID) == -1)
    {
        moveSliderTo(maxPosn);
        myTurn = true;
    }
}

void moveSliderTo(int newPosition) {
    while (analogRead(potPin) < newPosition) {
        digitalWrite(motorPin, HIGH);
    };
    digitalWrite(motorPin, LOW);
}
```

The original of this file has some extra commands commented out, that can be used to debug the project if the events don't seem to be getting through. See the comments in the original program if you need to use them.

The program starts by defining the two pins to be used. A0 for the voltage output of the potentiometer, which will be 0V if the rope is fully pulled out and 3.3V if the rope is fully pulled in.

The boolean variable *myTurn* is used to keep track of whose turn it is to pull on the rope. If *myTurn* is set to *true* then it is this device's turn to have its rope pulled.

Both ends of this project both publish and subscribe to the same event, so the variable *thisID* is needed so that the device knows its own ID and can disregard its own publish events, only reacting to events coming from the other Photon/Core.

The constant *maxPosn* is the analog input reading at which the sliding pot is at the position where the rope is fully pulled in. This is set slightly lower than the theoretical maximum analog input value of 4095 to allow for any inaccuracy in the analog readings.

The second constant *minPosn* is equivalent to about three quarters of the way pulled in and this is the threshold at which a "pulled" event will be published.

The *setup* function makes the necessary subscription to “pulled” associating it with the function *remoteRopePulled*. It also calls the function *moveSliderTo* to position the slider at its fully pulled in position, ready to be pulled out.

The *loop* function reads the analog input to find the *newLocalPosition*. If this is less than the *minPosn* constant and its this device’s turn to move, then the “pulled” event is published with this device’s ID as its parameter.

In the situation where the rope has been pulled on the other Photon/Core the function *remoteRobePulled* will be called. This function will be supplied with the ID of the device where the rope was pulled, so that it can be compared with thisID, the ID of the receiving Photon by searching for the string of characters in thisID within the ID passed in data.

If the event has come from a remote Photon/Core, then the slider is pulled fully in and *myTurn* flipped over to *true*.

The function *moveSliderTo* handles all automated movement of the slider. In fact it can only pull the slider in, but then you can’t push rope, so that’s fine. The function takes the new position as a parameter and keeps power supplied to the motor until such time as the measured position is no longer less than the desired position.

Hardware

The breadboard layout for this project is shown in Figure 8-5.

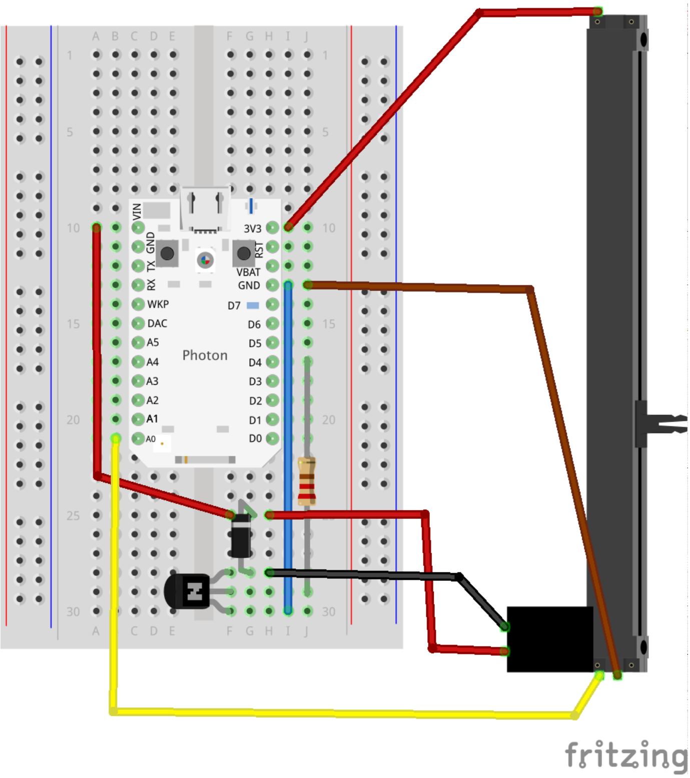

A B C D E F G H I J
1
5
10
15
20
25
30
VIN
GND
TX
RX
WKP
DAC
A5
A4
A3
A2
A1
A0
Photon
3V3
RST
VBAT
GND
D7
D6
D5
D4
D3
D2
D1
D0
fritzing

Figure 8-5. *Figure 8-5. The Magic Rope Breadboard Layout*

The motorized pots are actually stereo devices but we only need one channel for this project. This means that there are some pins that you do not need to connect leads to. Figure 8-6 shows the underside of the motorized pot, you can see the motor at the bottom right.

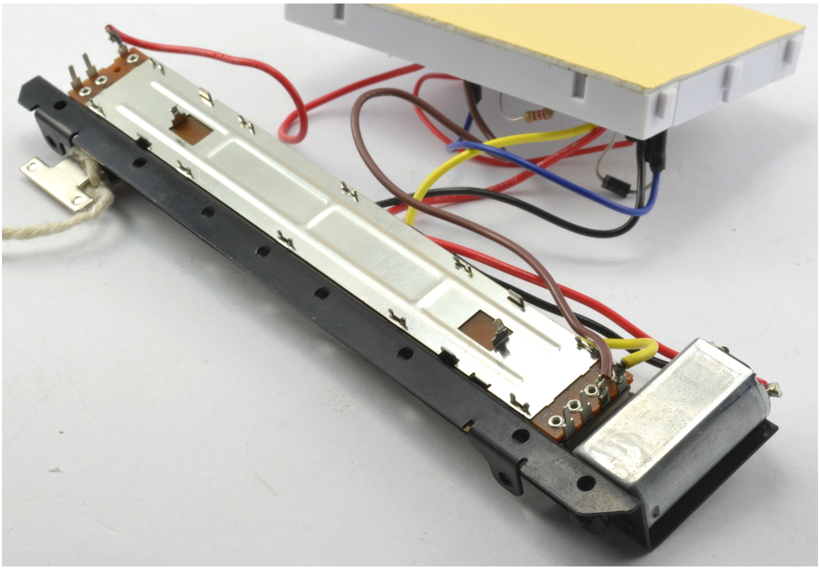


Figure 8-6. *Figure 8-6. The Underside of the Motorized Pot*

Before assembling the breadboard, you will need to solder some wires to the motorized pot legs. The motor leads are easily identified, and if you are using the same motorized pot as me, attach a red lead to the bottom-most motor lead (as shown in Figure 8-6) and a black lead to the other motor lead. All leads need to be about 6 inches long to comfortably reach the breadboard.

At the far end of the motor, attach a red lead to the right-most lead. This is the lead that will go to 3.3V on the breadboard. Attach a yellow or orange lead to the right-most pin at the motor end of the pot. This lead is the slider of the pot that will connect to A0 on the Photon/Core. Finally, connect a brown or blue lead next to this yellow lead. This will connect to GND on the breadboard.

Transistors

The motors of the motorized pots use upwards of 100mA which is far too much for a Photon/Core digital output which has an upper limit of about 20mA. To allow the digital output to turn the motor on and off a transistor is used.

Figure 8-7 shows the schematic diagram for the project.

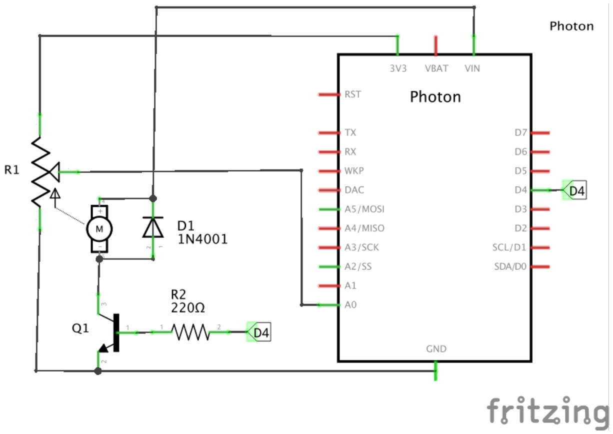

Photon
3V3
VBAT
VIN
Photon
RST
TX
RX
WKP
DAC
A5/MOSI
A4/MISO
A3/SCK
A2/SS
A1
A0
D7
D6
D5
D4
D3
D2
SCL/D1
SDA/D0
D4
R1
M
D1
1N4001
R2
220Ω
Q1
D4
GND
fritzing

Figure 8-7. *Figure 8-7. Using a Transistor to Control a Motor*

You can think of the transistor (Q1) as a kind of digital switch. That can use a small current to control a much bigger current.

When D4 is HIGH, a small current flows through R2 and the transistor to GND which stimulates a much higher current to flow from Vin (5V) through the motor, through the transistor and to GND.

Finally connect everything up as shown in Figure 8-5 paying special attention to the transistor and diode, to make sure they are the right way around. The diode has a stripe at one end that should be towards the top of the breadboard and the transistor has one curved side.

Driving motors can result in voltage spikes and the diode protects the Photon/Core from accidental damage resulting from these spikes.

Using the Project

To use the project, power up both ends of the magic rope. After the Photon/Core has finished starting up (flashing green light) the motor should activate on both ends of the project, pulling the slide up to one end of the track.

Pull on one of the strings and then the other. When you pull on the second string the first string should be pulled back automatically.

You could find a nice wooden box for this project, drilling a hole for the string to emerge at one end and a hole for the USB lead at the other.

Summary

This is the final chapter containing projects. In the next chapter various more advanced topics will be covered.

9/Advanced Photon

In this chapter you will learn about some of the more advanced features of the Photon and Core.

Configuring a Photon Using USB

Although the Tinker App is a very convenient way of setting up a new Photon/Core, it is not the only way of telling a new device your WiFi credentials. The USB socket on a Photon or Core is not just for supplying it with power, it can also be used to communicate with your computer over USB.

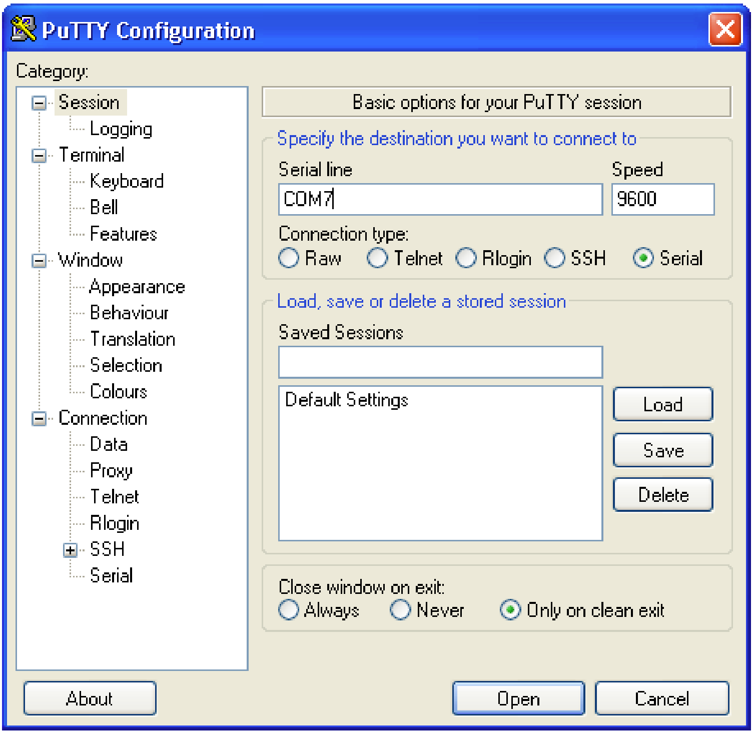
You can pass WiFi credentials to your Photon/Core using the USB connection and a serial communication program. If you have a Mac or Linux computer, then you already have a built-in utility (called *screen*) for this kind of serial communication over USB.

Windows Users

If you are a Windows user, you will need to install Putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/) and a USB driver for the Photon/Core (https://s3.amazonaws.com/spark-website/Spark.zip).

Extract the driver archive (Spark.zip) to your Desktop, you will need to point the Found New Hardware wizard at this folder later. Plug the Photon/Core in and when prompted for a driver, navigate to the folder you just downloaded.

To use Putty to communicate over USB with your Photon/Core, start Putty up and then in the row of radio buttons for "Connection type" select the option "Serial" (Figure 9-1).


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
Basic options for your PuTTY session
Specify the destination you want to connect to
Load, save or delete a stored session
Load
Save
Delete
About
Open
Cancel

Serial line: COM7

Speed: 9600

Connection type: ○ Raw ○ Telnet ○ Rlogin ○ SSH ● Serial

Saved Sessions: Default Settings

Close window on exit: ○ Always ○ Never ● Only on clean exit

Figure 9-1. *Figure 9-1. Setting-up Putty*

The Photon/Core will probably be allocated to COM7

If you are using Linux or a Mac, open a terminal window and issue the command:

```
$ screen screen /dev/cu.usbmodem1451 9600
```

When selecting the device (i.e., cu.usbmodem1451) the number on the end may be different for you, so press TAB after you have typed cu.usbmodem and your device should be autocompleted.

Whether you are using Putty or screen, you should now have an empty screen waiting for a command. Press the “i” key and you should see a message like this:

```
Your core id is 54ff6f065572524851401167
```

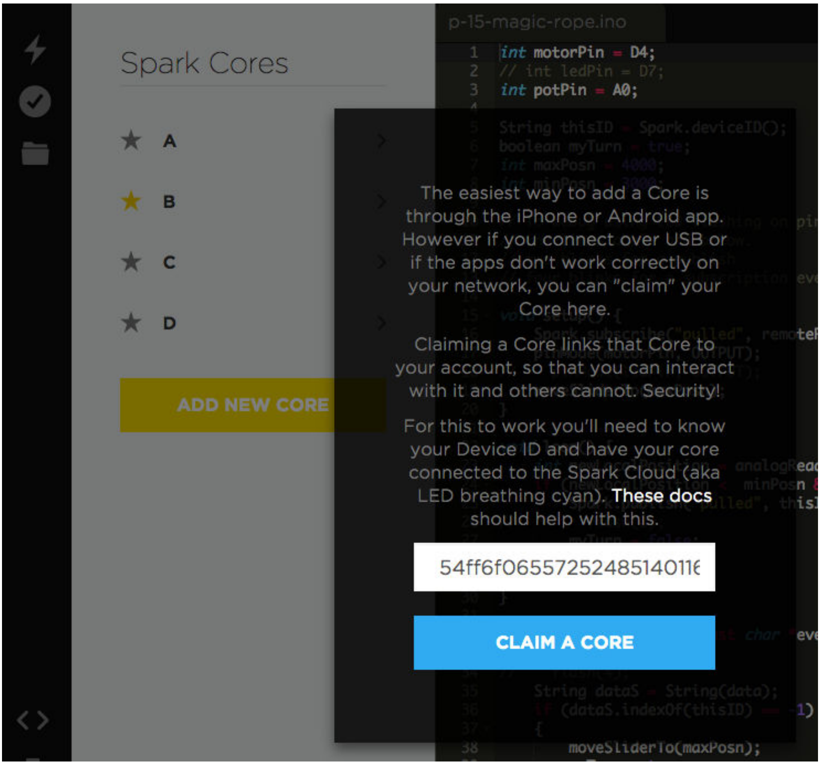
Make a note of this number, this is the ID for your Core. You will need it later when you come to manually claim your Core.

The other command that you can send to the Photon/Core is “w." This lets you tell the Photon/Core what your wireless network is called and its password. So, press the “w” key and you will be prompted for the SSID, security type (probably WPA2) and then your password.

```
SSID: mymobilenetwork
Security 0=unsecured, 1=WEP, 2=WPA, 3=WPA2: 3
Password: mypassword
Thanks! Wait about 7 seconds while I save those credentials...

Awesome. Now we'll connect!

If you see a pulsing cyan light, your Spark Core
has connected to the Cloud and is ready to go!

If your LED flashes red or you encounter any other problems,
visit https://www.spark.io/support to debug.

    Spark <3 you!
```

If all is well, your Photon/Core should reboot itself and then connect itself to the Spark Cloud indicated by a rhythmic cyan “breathing” of the RGB LED.

Although your core is now connected to the Spark Cloud, it does not know who it belongs to, so you need to “claim” it by logging into Spark.io and clicking on the “Cores” icon (it looks like a target). Then click the button ADD NEW CORE and paste in the long number you copied earlier after issuing the “i” command (Figure 9-2). Then click CLAIM A CORE. You will then be prompted for a name for the Photon/Core.



```
int motorPin = D4;
// int ledPin = D7;
int potPin = A0;
```

The easiest way to add a Core is through the iPhone or Android app. However if you connect over USB or if the apps don't work correctly on your network, you can "claim" your Core here.

Claiming a Core links that Core to your account, so that you can interact with it and others cannot. Security!

For this to work you'll need to know your Device ID and have your core connected to the Spark Cloud (aka LED breathing cyan). **These docs** should help with this.

54ff6f0655725248514011

Figure 9-2. *Figure 9-2. Configuring a Photon / Core Over USB*

You should now be able to use your Photon/Core as normal.

Factory Reset

From time to time, your Photon/Core will misbehave and need resetting. A simple press of the reset button or powering it on and off will restart the device and cure many problems; however, if the problem is something in your code, then it may stop you flashing replacement code onto the app. This is especially common if you are experimenting with control of the WiFi or making the device sleep to save power.

When all else fails, you can perform a full factory reset. To do this, press and hold down both buttons on the device and then release the Reset button. There will be some yellow flashing and then eventually it will start to flash white. When it starts flashing white, you can release the Setup (Mode on the Core) button and wait while the device does a full factory reset.

Eventually the white flashing will end and your device should be back to its original state. You will now have to tell it your WiFi credentials again.

Programming a Photon Using Spark Dev

I quite like the Web IDE and find it a convenient way to program and manage my devices. Spark also provides a desktop version of the IDE that you can download onto your computer. This is called Spark Dev and in many ways looks and operates in a very similar manner to the Web IDE.

At the time of writing, this tool is not fully off-line as it uses a web service to actually compile the app and flash it onto the device; however, this is a work in progress and full off-line programming of Photons and Cores over USB will be available in the future.

Debugging with the Serial Monitor

One problem with using an IoT device like the Photon or Core is that it can be tricky to work out what is happening when things go wrong with your code.

A common way to find out what is going on in a program is to add *trace* to your code. Trace is nothing more than program commands that print out some text somewhere so that you can see when the device runs a certain line of the program.

The problem with this on a device like the Photon or Core is that there is no screen on which the messages can appear. But what you can do is arrange for these messages to be sent over USB to a terminal application on your computer like Putty or the *screen* command that you used in "Configuring a Photon Using USB" on page 185.

As an example, try uploading the following app to your device. You can find the code in the file ch_09_Trace_Example in the PHOTON_BOOK library.

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    if (Serial.available()) {
        char ch = Serial.read();
        if (ch == '?') {
            Serial.print("Hello millis()=");
            Serial.println(millis());
        }
    }
}
```

Now open screen or Putty just as you did in #configuring-a-photon. Enter “?” in the terminal window, once its connected and you will see your device report the number of milliseconds since it last rebooted. Press ? again to repeat the process.

```
Hello millis()=101285
Hello millis()=107317
```

To start the communication the line Serial.println(9600) starts communication at the baud rate (communication speed) of 9600.

The loop function checks for an incoming communication from *screen* or Putty and if the character received is a “?” first a message string and then the result of calling millis() are printed over the serial connection.

The difference between *print* and *println* is just that *println* starts a new line at the end of the value it prints out, whereas *print* does not.

The Electron

A new Spark product is the Electron. At the time of writing, this is still at the stage of a Kickstarter project, but this device will function very like a Photon or Core, but instead of communicating using WiFi, the Electron communicates using a cellphonestype GSM modem. This means that the device can be truly mobile. For more news and updates on this interesting device, check back on the Spark.io website.

Power Management

WiFi uses quite a lot of power (generally up to say 200mA). To put that into perspective, this means that if you were powering your Photon or Core from a pack of four AA batteries, you might expect it to get about 10 hours use out of the batteries. With the WiFi turned off, you might easily get ten times as much time before your batteries are empty.

You can arrange for your device to put itself to sleep for period of time to increase the battery life of the project using the *sleep* command.

The sleep takes a number of seconds as its parameter and turns WiFi off for that period of time. For example, to put WiFi to sleep for 5 seconds, you would do:

```
    Spark.sleep(5);
```

There are other commands to explicitly turn WiFi on and off, but remember that if WiFi is off, there is no way to flash a replacement app on the device, so you may well need to do a factory reset. if things go wrong.

Check out the Spark documentation http://docs.spark.io/firmware for the latest documentation on these features.

Summary

There are many more advanced features of the Photon and Core that will need to wait for a more advanced book. This book is after all a getting started book. But, having mastered the basics check out the Spark documentation that is very thorough and well written. You will also find a helpful and informed community in the Spark forum at community.spark.io

A/Parts

Component Suppliers

Photon and Core boards are available from a wide range of suppliers and from Spark themselves.

You will also find kits, shields and other interesting Photon/Core compatible modules. Good suppliers for the hobbyist for such things are Adafruit and Sparkfun (no connection with Spark.io).

Parts

Most of the parts used in this book are available in the Maker's Kit available directly from Spark. This is by far the easiest way to source your components. All of the components are also available individually or as parts of other kits from other suppliers. The sections below list these parts and will give you some help finding them.

The codes next to each part match up to the codes used in the parts bins listed at the start of each project in the book.

Electronic Components

Even if you do not purchase the Spark Maker's Kit, the easiest way to buy a basic set of electronic components is to buy a general electronics starter kit. I have included Adafruit product codes for some of the less common components.

Code	Description	Source
C1	220Ω resistor	Kit
C2	Red LED	Kit
C3	Tactile push switch	Kit
C4	Piezo buzzer	Adafruit: 160
C5	1kΩ resistor	Kit

C6	Photo resistor (1kΩ)	Adafruit: 161
C7	DS18B20 temperature sensor (encapsulated)	eBay, Adafruit: 381
C8	100nF (0.1uF) capacitor	Kit
C9	4.7k resistor	Kit. Included in Adafruit: 381
C10	Transistor 2N3904	Adafruit: 756
C11	1N4001 Diode	Adafruit: 755

Modules and Shields

Code	Description	Source
M1	Relay Shield	Spark.io
M2	Photon Bot Robot Controller Shield	monkmakes.com (available May 2015)
M3	HC-SR04 ultrasonic rangefinder	eBay
M4	Slide Pot - Motorized (10k Linear Taper)	sparkfun.com 10976

Hardware and Connectors

Code	Description	Source
H1	DC jack to screw adapter (female)	Adafruit: 368
H2	DC jack to screw adapter (male)	Adafruit: 369
H3	Assorted multi-core hookup wire	Adafruit: 1311
H4	Male-to-male jumper wires	Adafruit: 758
H5	Half-sized breadboard	Adafruit: 64
H6	Rover robot chassis kit (6V)	eBay. Sparkfun: 12866
H7	Male 0.1inch header pins	Adafruit: 392

Other

Code	Description	Source
Q1	12V 1A Power supply with 2.1mm barrel jack	Adafruit: 798
Q2	12V DC electric bell	Hardware store

B/Photon LED Codes

The RGB LED of a Photon uses different colors and rates of flashing to indicate what is going on inside the device. After a while, it almost feels like the Photon is demonstrating its current mood by whats going on in its LED.

Reset Sequence

When you reset a Photon, it will go through the following sequence on the LED.

1. Flashing blue: Listening mode, waiting for network information.
2. Solid blue: Smart Config complete, network information found.
3. Flashing green: Connecting to local Wi-Fi network.
4. Flashing cyan: Connecting to Spark Cloud.
5. High-speed flashing cyan: Spark Cloud handshake.
6. Slow breathing cyan: Successfully connected to Spark Cloud.

Other Status Codes

- Flashing yellow: Bootloader mode, waiting for new code via USB or JTAG. JTAG is a type of specialized programming hardware
- White pulse: Start-up, the Core was powered on or reset.
- Flashing white: Factory Reset initiated.
- Solid white: Factory Reset complete; rebooting.
- Flashing magenta: Updating firmware.

- Solid magenta: May have lost connection to the Spark Cloud. Pressing the Reset (RST) button will attempt the update again.

Error Codes

Generally red flashing indicates a problem.

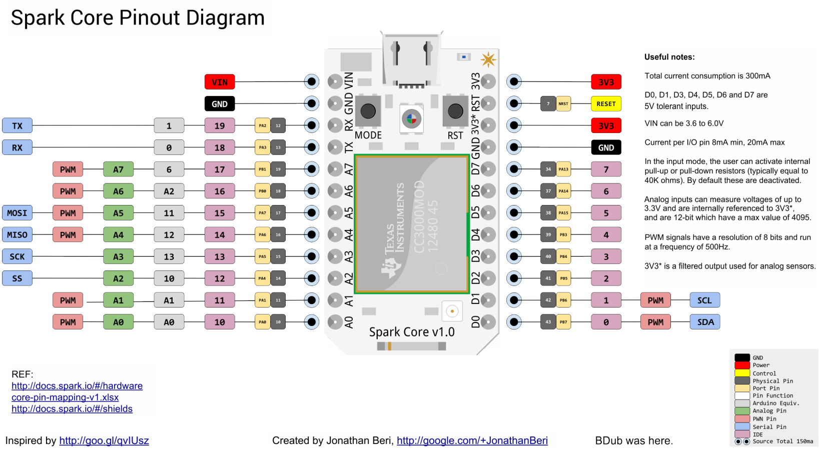
- Two red flashes: Connection failure due to bad internet connection.
- Three red flashes: The Cloud is inaccessible, but the internet connection is fine.
- Four red flashes: The Cloud was reached but the secure handshake failed.
- Flashing yellow/red: Bad credentials for the Spark Cloud.

You can find out more about resolving these kinds of problem at Spark's support page (http://support.spark.io/).

C/Photon and Core Pinouts

These diagrams summarize the uses that each of the various pins can be put to on the Core and Photon devices.

Core


Spark Core Pinout Diagram
VIN
GND
TX 1 19 PA2 12 RX
RX 0 18 PA3 13 TX
PWM A7 6 17 PB1 19 A7
PWM A6 A2 16 PB0 18 A6
MOSI PWM A5 11 15 PA7 17 A5
MISO PWM A4 12 14 PA6 16 A4
SCK A3 13 13 PA5 15 A3
SS A2 10 12 PA4 14 A2
PWM A1 A1 11 PA1 11 A1
PWM A0 A0 10 PA0 10 A0
3V3 3V3
RST 7 NRST RESET
3V3* 3V3
GND GND
D7 34 PA13 7
D6 37 PA14 6
D5 38 PA15 5
D4 39 PB3 4
D3 40 PB4 3
D2 41 PB5 2
D1 42 PB6 1 PWM SCL
D0 43 PB7 0 PWM SDA
MODE
RST
TEXAS INSTRUMENTS
CC3000MOD
1248045
Spark Core v1.0
GND
Power
Control
Physical Pin
Port Pin
Pin Function
Arduino Equiv.
Analog Pin
PWN Pin
Serial Pin
IDE
Source Total 150ma

Useful notes:

Total current consumption is 300mA

D0, D1, D3, D4, D5, D6 and D7 are 5V tolerant inputs.

VIN can be 3.6 to 6.0V

Current per I/O pin 8mA min, 20mA max

In the input mode, the user can activate internal pull-up or pull-down resistors (typically equal to 40K ohms). By default these are deactivated.

Analog inputs can measure voltages of up to 3.3V and are internally referenced to 3V3*, and are 12-bit which have a max value of 4095.

PWM signals have a resolution of 8 bits and run at a frequency of 500Hz.

3V3* is a filtered output used for analog sensors.

REF:
http://docs.spark.io/#/hardware
core-pin-mapping-v1.xlsx
http://docs.spark.io/#/shields

Inspired by http://goo.gl/qvlUsz

Created by Jonathan Beri, http://google.com/+JonathanBeri

BDub was here.

Figure C-1. *The Core Pinout*

Photon

The final image is not yet available.